

Indice

1	Introduction (lezione 1 Virdis)	1
1.1	Concetti chiave	1
2	complessità dei programmi (lezione 2 Virdis)	4
2.1	complessità dei programmi iterativi	7
3	funzioni e classi modello (lezione 1 Ducange)	8
3.1	funzioni	9
3.2	classi	16
4	Esercizi (lezione 3 Virdis)	17
4.1	Ricerca Binaria (versione iterativa)	17
5	Lezione 4 Virdis	18
5.1	comandi base Unix	18
5.2	Ordinamento: insertion sort	19
5.3	Debug	19
5.4	Standard Template Library	19
6	Derivazione (Lezione 2 Ducange)	21
6.1	Regole di visibilità	25
6.1.1	Regole di visibilità con i puntatori	26
6.1.2	Regole di visibilità con le funzioni membro	26
6.2	specificatori di accesso	27
6.3	costruzione degli oggetti	28
6.4	membri statici	32
7	Funzioni virtuali, classi astratte e Polimorfismo (Lezione 3 Ducange)	34
7.1	Funzioni virtuali	34
7.1.1	Funzioni virtuali nella gerarchia	36
7.1.2	Distruttori virtuali	36
7.2	Classi astratte e polimorfismo	37
7.3	Gestione delle eccezioni	40

8 Programmi ricorsivi (lezione 5 virdis)	46
8.1 QuickSort	47
8.2 Ricerca in un insieme	47
9 MergeSort, ordinamento STL e gestione liste (Lezione 6 Virdis)	49
9.1 Merge Sort	49
9.1.1 nozioni aggiuntive	49
9.2 Complessità	50
9.3 Compiler Flags	50
9.4 STL sort	50
9.5 Valgrind	50
10 Torre di Hanoi, ricerca in un insieme e metodo "dividi et impera"	51
10.1 torre di Hanoi (Lezione 7 Virdis)	51
10.2 ricerca in un insieme	52
10.3 Dividi et impera	53
10.3.1 Classificazione di alcune relazioni di ricorrenza (non lineari)	54
10.3.2 Classificazione di alcune relazioni di ricorrenza (lineari)	55
10.4 Algoritmi di teoria dei numeri	56
11 Serie di Fibonacci e Merge sort (Lezione 8 Virdis)	57
12 Alberi binari (Lezione 4 Ducange)	58
12.1 Intro	58
12.2 Ricorsione su alberi binari	59
12.3 Visite di alberi binari	59
12.3.1 Memorizzazione in lista multipla	59
12.3.2 Visita anticipata (preorder)	59
12.3.3 Visita differita (postorder)	60
12.3.4 Visita simmetrica (inorder)	60
12.3.5 Complessità delle visite	60
12.4 Bilanciamento degli alberi	61
12.5 Complessità delle visite nel numero dei livelli	61
12.6 Funzioni su alberi	61
12.6.1 conta i nodi	61
12.6.2 conta le foglie	62
12.6.3 Cerca un'etichetta	62
12.6.4 Cancella tutto l'albero	62
12.6.5 Inserisci un nodo	62
12.7 Classe Bin Tree	63

13 Alberi generici (Lezione 5 Ducange)	65
13.1 intro	65
13.1.1 Differenza con alberi binari	66
13.1.2 Visite	66
13.1.3 memorizzazione	66
13.1.4 corrispondenza tra visite	67
13.1.5 esempi	68
13.2 Alberi binari di ricerca	69
13.2.1 Proprietà e operazioni	69
14 heap (Lezione 6 Ducange)	72
14.1 Classe Heap e operazioni	72
14.1.1 Inserimento	73
14.1.2 Estrazione	74
14.2 Algoritmo Heapsort	75
14.2.1	75
14.2.2 Trasforma l'array in uno heap (buildHeap)	76
14.2.3 Extract modificata	76
14.2.4 risultato finale	77
15 Alberi Binari di Ricerca, Gestione Stringhe, Progettazione (Lezione 9 Virdis)	78
15.1 Stringhe	79
16 Limiti Inferiori, Alberi di Decisione, Algoritmi di ordinamento (Lezione 7 Ducange)	81
16.1 Limiti inferiori	81
16.2 Alberi di decisione	81
16.3 Algoritmi di ordinamento	83
16.3.1 Counting sort	84
16.3.2 Radix Sort	85
17 Heap (Lezione 10 Virdis)	87
17.1 Classe heap	88
17.1.1 Funzioni relative all'immagine 17.1:	89
17.1.2 Stampa con la grafica ganza	90
17.1.3 heap property	91
17.1.4 Funzione heapify	91
17.1.5 Funzione build heap (MaxHeap)	92
17.1.6 HeapSort	93
17.1.7 Heap STL	93

18 Hash (Lezione 8 Ducange)	94
18.1 Metodo hash ad indirizzamento aperto (non si fa uso di puntatori(?))	95
18.1.1 Funzione hash modulare	95
18.1.2 Tempo medio di ricerca per l'indirizzamento aperto	98
18.1.3 Stima del numero medio di accessi	98
18.1.4 Problemi con l'indirizzamento aperto	99
18.2 Metodo di concatenazione	99
18.3 Dizionari (tabelle)	99
19 Programmazione dinamica e Algoritmi Greedy (Lezione 9 Ducange)	101
19.1 Più lunga sottosequenza comune	102
19.1.1 Estrazione di una PLSC	104
19.2 Algoritmi greedy (Avido/goloso)	105
19.2.1 codici di compressione	105
19.2.2 Algoritmo di Huffman	107
20 Grafi (lezione 10 Ducange)	109
20.1 Grafi orientati	109
20.1.1 rappresentazione in memoria dei grafi	110
20.1.2 visita in profondità	112
20.1.3 classe per i grafi	112
20.2 Grafi non orientati	113
20.2.1 Rappresentazione in memoria	113
20.3 Multi grafi orientati	114
20.4 Minimo albero di copertura	115
20.4.1 algoritmo di Kruskal per trovare il minimo albero di copertura	115
21 Grafi 2 (Lezione 11 Ducange)	117
21.1 Algoritmo di Dijkstra	117
21.2 Algoritmo PageRank	119
21.3 Graph Database	120
22 NP-Completezza (Lezione 12 Ducange)	122
22.1 Problemi difficili	122
22.1.1 Zaino	122
22.1.2 commesso viaggiatore	123
22.1.3 Cammino e ciclo Hamiltoniano	123
22.1.4 soddisfattibilità di una formula logica	124
22.1.5 Problema del ciclo euleriano	125
22.2 NP-completezza	126
22.2.1 Algoritmi nondeterministici	127

22.2.2	Un algoritmo nondeterministico di ricerca in array	128
22.2.3	Un algoritmo nondeterministico di ordinamento	128
22.2.4	Relazione fra determinismo e nondeterminismo	128
22.2.5	Riducibilità	129
22.2.6	Teorema di Cook	130
22.2.7	NP-completezza	130
22.2.8	Problema della fattorizzazione di un numero	132
23	Hashing (Lezione 11 Virdis)	135
23.1	Hashing	135
23.1.1	Simple hash table (esempio 1)	135
23.1.2	collisioni	136
23.1.3	Hashing stringhe	138
23.2	fine leizone	139
23.3	STL map	139
24	Algoritmi di sorting	140
24.1	selection sort	140
24.2	Bubble sort	140
24.3	insertion sort	141
24.4	Quick sort	141
24.5	Merge sort	142
24.6	STL sort	143
24.7	HeapSort	143
24.8	CountingSort	143
24.9	RadixSort	143

List of Theorems

1	Definizione (Algoritmo)	1
2	Definizione (complessità di un algoritmo)	2
3	Definizione (profiling)	2
4	Definizione (limite asintotico superiore: Notazione O grande)	4
5	Definizione (teorema)	5
6	Definizione (Notazione Ω omega grande (limite asintotico inferiore))	5
7	Definizione (Notazione Θ theta grande (limite asintotico stretto))	6
8	Definizione (classe)	8
9	Definizione (oggetto)	8
10	Definizione (Meta-Programmazione)	8
11	Definizione (Funzioni modello: costrutto template)	9
12	Definizione (dichiarazione e definizione di template)	16
13	Definizione (dichiarazione e definizione classe modello)	16
14	Definizione (Insertion sort)	19
15	Definizione (vvector)	19
16	Definizione (funzione pushback)	19
17	Definizione (compatibilità fra tipi (puntatori))	23
18	Definizione	27
19	Definizione	28
20	Definizione	28
21	Definizione (distruzione)	31
22	Definizione (classe astratta)	37
23	Definizione (funzione virtuale pura)	37
24	Definizione (Eccezioni)	40
25	Definizione (Eccezioni: costrutto sintattico)	40
26	Definizione (Regole sulla ricorsività)	46
27	Definizione (lista)	46

28	Definizione (Complessità dei programmi ricorsivi)	47
29	Definizione (Ricerca lineare ricorsiva)	47
30	Definizione (ricerca binaria ricorsiva)	47
31	Definizione (Albero binario)	58
32	Definizione (Albero binario bilanciato)	61
33	Definizione (Alberi binari quasi bilanciati)	61
34	Definizione (Alberi pienamente binari)	61
35	Definizione (Albero generico)	65
36	Definizione (Albero binario di ricerca)	69
37	Definizione (proprietà)	69
38	Definizione (Operazioni)	69
39	Definizione (heap)	72
40	Definizione (Algoritmo di ordinamento Heapsort)	75
41	Definizione (Limite inferiore)	81
42	Definizione (Albero di decisione)	81
43	Definizione (Funzione hash)	94
44	Definizione (Funzione hash modulare)	95
45	Definizione (Legge di scansione lineare)	95
46	Definizione (Programmazione dinamica)	101
47	Definizione (Algoritmo di Huffman)	107
48	Definizione (Grafi orientati)	109
49	Definizione (arco)	109
50	Definizione (cammino)	109
51	Definizione (ciclo)	110
52	Definizione (liste di adiacenza)	110
53	Definizione (matrici di adiacenza)	110
54	Definizione (grafo non orientato)	113
55	Definizione (Multi-Grafo)	114
56	Definizione (algoritmo di Dijkstra)	117
57	Definizione (SAT)	124
58	Definizione (Teorema di Eulero)	126
59	Definizione (Teoria della NP-completezza)	126
60	Definizione (problemi decisionali)	127
61	Definizione (Algoritmi nondeterministici)	127
62	Definizione (Teorema di Cook)	130
63	Definizione (NP-completezza)	130

64	Definizione (selection sort)	140
65	Definizione (Bubble sort)	140
66	Definizione (Insertion sort)	141
67	Definizione (Quick sort)	141
68	Definizione (Merge sort)	142

Algoritmi e Strutture Dati

Francesco Bonistalli

March 2022

Capitolo 1

Introduction (lezione 1 Virdis)

Definizione 1 Algoritmo: procedimento che descrive una sequenza di passi ben definiti finalizzato a risolvere un dato problema (computazionale = "relativo al calcolo").

- insieme finito di istruzioni teso a risolvere un problema
- Input e Output
- ogni istruzione deve essere ben definita ed eseguibile in un tempo finito da un agente di calcolo
- E' possibile utilizzare una memoria per i risultati intermedi

Un algoritmo può essere visto come l'essenza computazionale di un programma, nel senso che fornisce il procedimento per giungere alla soluzione di un dato problema di calcolo.

Algoritmo ≠ Programma

- programma è la codifica (in un linguaggio di programmazione) di un algoritmo
- un algoritmo può essere visto come un programma distillato da dettagli riguardanti il linguaggio di programmazione, ambiente di sviluppo, sistema operativo
- Algoritmo è un concetto autonomo da quello di programma

1.1 Concetti chiave

(esempi relativi al problema "Trovare tutti i numeri primi fino a un dato numero n")

- **problema:** individuare i numeri primi minori di n

- **istanza:** i numeri da 0 a n
- **dimensione dell'istanza:** il valore di n
- **modello di calcolo:** insieme di caselle contenente i numeri
- **algoritmo:** strategia di calcolo dei numeri primi. La descrizione deve essere “comprensibile” e “compatta”. Deve descrivere la sequenza di operazioni sul modello di calcolo eseguite per una generica istanza
- **correttezza dell'algoritmo:** la strategia di calcolo deve funzionare per una generica istanza, ovvero indipendentemente da quante caselle ci sono.
- **complessità temporale:** numero di operazioni che esegue prima di individuare TUTTI i numeri primi dentro l'istanza. Dipende dalla dimensione dell'istanza e dall'istanza stessa.
- **complessità temporale nel caso peggiore:** numero massimo di operazioni che esegue su una istanza di una certa dimensione. E' una delimitazione superiore a quanto mi "costa" risolvere una generica istanza. Espressa come funzione della dimensione dell'istanza.
- **efficienza:** l'algoritmo deve fare poche operazioni, deve essere cioè veloce. Ma veloce rispetto a che? quando si può dire che un algoritmo è veloce?

Definizione 2 complessità di un algoritmo: funzione (sempre positiva) che associa alla dimensione del problema il costo della sua risoluzione

- **costo:** tempo, spazio (memoria), ...
- **dimensione:** dipende dai dati

Per confrontare due algoritmi si confrontano le relative funzioni di complessità

Differenza tra complessità e profiling

Definizione 3 profiling: tecnica di analisi di un programma basata sulla misurazione di indici di prestazione del programma stesso (e.g., utilizzo memoria, numero di chiamate di funzione, etc.) Questa misura dipende da fattori specifici dell'ambiente di esecuzione (calcolatore, sistema operativo, compilatore, linguaggio di programmazione) e dall'istanza considerata.

Il concetto di complessità deriva quindi dalla necessità di trovare un metodo di calcolo della complessità che misuri l'efficienza come proprietà dell'algoritmo, che quindi astragga dal computer su cui l'algoritmo è eseguito e dal linguaggio in cui l'algoritmo è scritto

Il tempo di esecuzione lo misuriamo quindi in funzione della dimensione n delle istanze, tuttavia istanze diverse, a parità di dimensione, potrebbero richiedere tempo di esecuzione diverso: distinguiamo quindi l'analisi in caso peggiore, medio e migliore.

Capitolo 2

complessità dei programmi (lezione 2 Virdis)

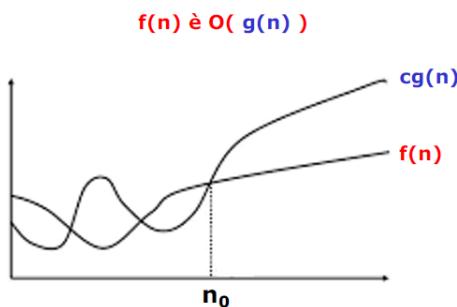
L'efficienza deve essere misurata indipendentemente anche da specifiche dimensioni dei dati: la funzione della complessità deve essere analizzata nel suo comportamento asintotico

Definizione 4 limite asintotico superiore: Notazione **O grande:** $f(n)$ è di ordine $O(g(n))$ se esistono un intero n_0 ed una costante $c > 0$ tali che

$$\forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

Notazione:

- $f(n)$ è di ordine $O(g(n))$
- $f(n) \in O(g(n))$
- $f(n) \in O(g(n))$



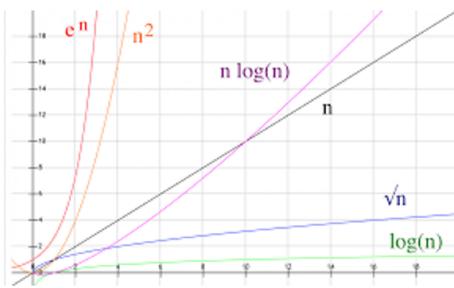
Regole sulla complessità computazionale:

Dato $O(f(n)) =$ insieme delle funzioni di ordine $O(f(n))$

- **regola dei fattori costanti** Per ogni costante positiva k , $O(f(n)) = O(kf(n))$.
- **regola della somma** Se $f(n)$ è $O(g(n))$, allora $f(n)+g(n)$ è $O(g(n))$.
- **regola del prodotto** Se $f(n)$ è $O(f_1(n))$ e $g(n)$ è $O(g_1(n))$, allora $f(n)g(n)$ è $O(f_1(n)g_1(n))$.
- Se $f(n)$ è $O(g(n))$ e $g(n)$ è $O(h(n))$, allora $f(n)$ è $O(h(n))$
- per ogni costante k , k è $O(1)$ per ogni costante k , k è $O(1)$
- per $m \leq p$, nm è $O(np)$
- Un polinomio di grado m è $O(n^m)$

Classi di Complessità

$O(1)$	costante
$O(\log n)$	logaritmica ($\log_a n = \log_b n \log_b a$)
$O(n)$	lineare
$O(n \log n)$	nlogn
$O(n^2)$	quadratica
$O(n^3)$	cubica
..	
$O(n^p)$	polinomiale
$O(2^n)$	esponenziale
$O(n^n)$	esponenziale



Definizione 5 teorema:

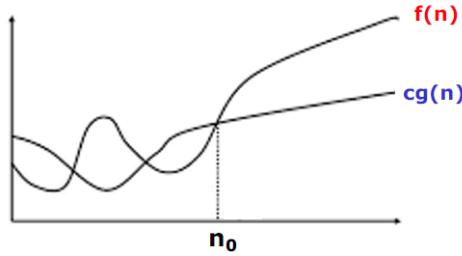
$$\forall k, n^k \in O(a^n), \forall a > 1$$

Una qualsiasi funzione polinomiale ha minore complessità di una qualsiasi funzione esponenziale

Definizione 6 Notazione Ω omega grande (limite asintotico inferiore): $f(n) \geq (g(n))$ se esistono un intero n_0 ed una costante $c > 0$ tali che

$$\forall n \geq n_0 : f(n) \geq c \cdot g(n)$$

$f(n)$ è $\Omega(g(n))$

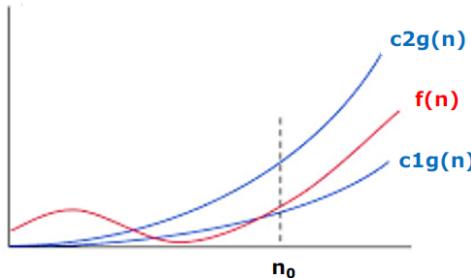


Definizione 7 Notazione Θ theta grande (limite asintotico stretto):
 $f(n)$ è $\Theta(g(n))$ se $f(n)$ è $O(g(n))$ e $f(n)$ è $\Omega(g(n))$.

$f(n)$ è $\Theta(g(n))$ quando f e g hanno lo stesso ordine di complessità. Definizione alternativa: $f(n)$ è $\Theta(g(n))$ se esistono un intero n_0 e due costanti $c_1, c_2 > 0$ tali che

$$\forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

$f(n)$ è $\Theta(g(n))$



risultati

- a) $f(n)$ è $O(g(n))$ se e solo se $g(n)$ è $\Omega(f(n))$
- b) se $f(n)$ è $\Theta(g(n))$ allora $g(n)$ è $\Theta(f(n))$
- c) Per Ω e Θ valgono le regole dei fattori costanti, del prodotto e della somma
- d) un polinomio di grado m è $\Theta(n^m)$
- e) O, Ω e Θ sono relazioni transitive

2.1 complessità dei programmi iterativi

```
void exchange( int& x, int& y) {  
    int temp = x;  
    x = y;  
    . . .  
    y = temp;  
}  
  
void selectionSort(int A[ ], int n) {  
    for (int i=0; i< n-1; i++) {  
        int min= i;  
        for (int j=i+1; j< n; j++)  
            if (A[ j ] < A[min]) min=j;  
        exchange(A[i], A[min]);  
    }  
}
```

Selection sort

O(n²)

Bubblesort

```
void bubbleSort(int A[], int n) {  
    for (int i=0; i < n-1; i++)  
        for (int j=n-1; j >= i+1; j--)  
            if (A[j] < A[j-1]) exchange(A[j], A[j-1]);  
}
```

O(n²)

numero di scambi = **O(n²)**

con selectionSort numero di scambi = **O(n)**

esempi vari vedi fine pdf (quite usless)

Capitolo 3

funzioni e classi modello (lezione 1 Ducange)

Definizione 8 classe: Descrizione di un gruppo di oggetti con proprietà (attributi), comportamento (operazioni), relazioni e semantica comuni.

La classe è l'astrazione che:

- enfatizza caratteristiche rilevanti
- sopprime le altre caratteristiche

Definizione 9 oggetto: manifestazione concreta di un'astrazione.

Istanza di una classe.

Vantaggi della programmazione a oggetti: encapsulamento, decomposizione, riuso, manutenzione, affidabilità.

Definizione 10 Meta-Programmazione: Possibilità di applicare lo stesso codice a tipi diversi, parametrizzando i tipi utilizzati:

Indipendenza degli algoritmi dai dati a cui si applicano: per esempio, un algoritmo di ordinamento può essere scritto una sola volta, qualunque sia il tipo dei dati da ordinare.

3.1 funzioni

```
int i_max(int x, int y) {
    return (x>y) ? x : y;
}
double d_max(double x, double y) {
    return (x>y) ? x : y;
}

void main() {
    int b; double c;
    // ...
    a= i_max(3,b);
    d=d_max(3.6,c);
}
```

Le due funzioni hanno la stessa definizione con tipi diversi

Figura 3.1: esempio di funzioni modello

Definizione 11 Funzioni modello: costrutto template: .

```
#include <iostream>
template <class tipo>
tipo max(tipo x, tipo y){
    return (x > y) ? x : y;
}
int main() {
    int b;
    double c;
    //...
    b = max(3, b);
    //tipo = int max<int>(int, int)
    c = max(3.6, c);
    //tipo = double max<double>(double, double)
}
```

Funzioni modello: argomenti impliciti

```
template<class tipo>
tipo max(tipo x, tipo y) .....

void main() {
    int b=2; double c=6.0, d; int array[2]={3,4};

    cout << max(array[0],b); // OK: int max<int>(int,int)

    d = max(3.6,c);
        // OK: double max<double>(double, double)

    b = max(3.6,c);
        // OK: double max<double>(double, double) e conversione

    // d = max(3,c); errore: non si deduce il tipo:
                    // 3 e' intero, c e' double
}
```

I tipi devono essere deducibili dalla chiamata

Esempio di funzione modello (I)

```
template<class tipo>
void primo ( tipo *x ) {
    tipo y= x[0];
    cout << y << endl;
};

void main() {
    int array1[2]={3,4};
    double array2[2]={3.5,4.8};

    primo(array1);
        // 3  tipo=int void primo<int>(int*)

    primo(array2);
        // 3.5 tipo=double void primo<double>(double*)
}
```

Esempi di funzioni modello (II)

```
template<class tipo>
void primo ( tipo x ) {
    cout << x[0] << endl;
};

void main() {
    int array1[2]={3,4};
    double array2[2]={3.5,4.8};

    primo(array1);

    // 3  tipo=int*  void primo<int*>(int*)
    primo(array2);

    // 3.5  tipo=double*  void primo<double*>(double*)
}
```

Esempi di funzioni modello (III)

```
template<class tipo>
void primo ( tipo x ) {
    tipo y= x[0];
    cout << y << endl;
};

void main() {
    int array1[2]={3,4},
        primo(array1);

    //  tipo=int*  void primo<int*>(int*)
    //errore non si può convertire un int in int*!!!
}


```

funzioni modello con più parametri

```
template<class tipo1, class tipo2>
tipo1 max(tipo1 x, tipo2 y) {
    return (x>y) ? x : y;
}

void main() {
    int b=2; double c=6;

    cout << max(3,b); // int max<int,int>(int,int)
    // tipo1=int, tipo2= int

    b = max(3,c); // int max<int,double>(int,double)
    // tipo1=int, tipo2= double
}
```

funzioni modello con più parametri

```
template<class tipo1, class tipo2, class tipo3>
tipo1 nuovomax(tipo2 x, tipo3 y) {
    return (x>y) ? x : y;
}

void main() {
    int b; double c=6;

    b = nuovomax(3,c);
    // NO: tipo1=? , tipo2=int, tipo3=double
}
```

Funzioni modello: parametri esplicativi

```
template<class tipo>
tipo max(tipo x, tipo y) {
    return (x>y) ? x : y;
}

void main() {
    double d;
    cout << max<int>(3,5.5);
    // 5 max<int>(int,int); conversione del parametro

    cout << max<double>(3,5.5);
    // 5.5 max<double>(double,double) conversione
    // del parametro

    d= max<int>(3,5.5);
    // max<int>(int,int); conversione del valore
    // assegnato: 5

}
```

Funzioni modello: parametri esplicativi e impliciti

```
template<class tipo1, class tipo2, class tipo3>
tipo1 fun(tipo2 x, tipo3 y) {
    ....
}

Gli argomenti esplicativi sono indicati nell'ordine del template
fun<int>(9,8.8); // tipo1= int : int fun<int,int,double>

fun<int,double>(9,8.8); // tipo1=int, tipo2=double :
                           int fun<int,double,double>

fun<int,int,double>(...); // int fun<int,int,double>

fun(9,8); // errore tipo3=tipo2=int, tipo1?
```

Funzioni modello: parametri costanti

```
template<int n, double m >
void funzione(int x=n){
    double y=m;
    int array[n];
    ....
}

void main () {
    funzione<1+2,2>(8); // n=3, m=2 funzione<3,2>(int)

    funzione<2,2>(9); // n=2, m=2 funzione<2,2>(int)
}
```

I parametri costanti sono necessariamente esplicativi:
Le istanziazioni di n e m devono essere ESPRESSIONI COSTANTI

Funzioni modello: parametri costanti e no

```
template< int n, class T>
int gt(T x){
    return x>n;
}

void main(){
    cout << gt<50+6>(101);
    // 1 n=56, T=int int gt<56,int>(int)
    // risoluzione implicita di T

    cout << gt<8, double>(7);
    // 0 n=8, T=double int gt<8, double>(double)
    // risoluzione esplicita di T
```

Funzioni modello con variabili statiche

```
template<class tipo>
tipo maxT(tipo x, tipo y) {
    static int a; a++; cout << a << endl;
    return (x>y) ? x : y;
}

void main(){
    cout << maxT<int>(101,102) << endl;      // 1 102
    cout << maxT<int>(101,102)<< endl;      // 2 102
    cout << maxT<double>(101,102) << endl;   // 1 102
}
```

Ogni istanza della funzione ha la sua variabile statica

Definizione 12 dichiarazione e definizione di template: Una funzione modello non può essere compilata senza conoscere le chiamate: non si può fare una compilazione separata

```
// file templ.h
template<class tipo>
void boh(tipo x){
    // ... definizione
}
// file main
#include "templ.h"
void main() {
    //...
    boh(6);
    // ...
}
```

3.2 classi

Anche le classi possono essere definite come classi modello:

```
template<class tipo1, class tipo2, int n ...>
class obj {...
```

I parametri in questo caso sono sempre esplicativi.

Definizione 13 dichiarazione e definizione classe modello: .

```
// file stack.h
// contiene dichiarazioni e definizioni
template<class tipo>
class stack{
    // ..
public:
    ...
};
// definizioni
// file principale
#include "stack.h"
...
```

(esempi vari pdf da pag 28)

Capitolo 4

Esercizi (lezione 3 Virdis)

4.1 Ricerca Binaria (versione iterativa)

```
int binSearch_it(int A[], int x, int l, int r) (versione iterativa)
{
    int m = (l+r)/2;
    while( A[m] != x )
    {
        if(x < A[m])
            r = m-1;
        else // x > A[m]
            l = m+1;
        if(l>r)
            return -1;
        m = (l+r)/2;
    }
    return m;
}
```

Caso migliore = $O(1)$.

Caso peggiore: al passo i , array ha dimensione $\frac{n}{2^i}$. Quindi nel caso peggiore in cui $l = r$ si ha $\frac{n}{2^i} \rightarrow i = \log(n)$

Capitolo 5

Lezione 4 Virdis

5.1 comandi base Unix

Navigare tra le cartelle:

- ls = lista sottocartelle
- pwd = quale cartella sto controllando
- cd "cartella" = entrare nella cartella
- cd .. = torna indietro

Manipolazione file e cartelle:

- mkdir "cartella" = crea una directory
- touch "file" = crea file vuoto
- cp/mv "sorgente destinazione" = copia (copy, paste) / muovi (cut, paste)
- rm "file/cartella" = rimuovi
- rm -R "cartella" = rimuovi ricorsivo (rimuovi cartella e ricorsivamente il suo contenuto)

Compilare un singolo file C++:

- g++ [opzioni] -o "eseguibile sorgente.cpp"

Eseguire un programma:

- ./"eseguibile"

Redirezione da file:

- ./"eseguibile" < "file di input"

comando time:

time ./"eseguibile"

5.2 Ordinamento: insertion sort

Definizione 14 Insertion sort: mano e occhio: confronta, avanzandola ogni volta, la mano con gli elementi precedenti, posizionandola al posto giusto e spostando anch'essi.

```
void sortArray( int arr[] , int len )
{
    int mano = 0;
    int occhio = 0;
    for( int iter = 1 ; iter < len ; ++iter )
    {
        mano = arr[iter];
        occhio = iter-1;
        while( occhio >= 0 && arr[occhio] > mano )
        {
            arr[occhio+1] = arr[occhio];
            --occhio;
        }
        arr[occhio+1] = mano;
    }
}
```

5.3 Debug

5.4 Standard Template Library

Definizione 15 vvector: container di dati di tipo generico il cui contenuto è specificato tramite template

```
//esempio
vector<int> stlArray
```

Definizione 16 funzione pushback: aggiunge in coda un intero (nel caso dell'esempio precedente) contenuto nella variabile val; se non c'è posto "allunga" l'array di una casella (non esattamente una !spoiler!).

push_back(val)

Altre possibilità della libreria stl:

- fornisce una funzione per l'accesso randomico -> permette di accedere all'iesima locazione in memoria con costo O(1) ovvero costo costante
- ci permette di accedere all'indirizzo della prima locazione di memoria del vettore

- ci permette di avere degli iteratori (=indici) al primo e all'ultimo elemento del vettore: stlArray.begin() e stl.Array.end() (non sono per forza inizio e fine dell'array)
- permette di sapere a priori la dimensione dell'array (stl.Array.size())
- il vector permette tutto ciò allocando ogni volta che serve aumentare lo spazio 10 elementi; ogni volta ogni volta trasferisce tutto su un array con 10 posti in più: il puntatore al primo indirizzo cambia ogni volta

Capitolo 6

Derivazione (Lezione 2 Ducange)

La derivazione o ereditarietà consente di trasmettere un insieme di caratteristiche comuni da una classe base ad una derivata senza che ciò comporti una duplicazione del codice, offrendo allo stesso tempo l'opportunità di adattare o estendere il comportamento a casi d'uso specifici.

Attraversare i livelli della gerarchia dall'alto verso il basso significa spostarsi da un livello di astrazione generico ad altri sempre più specifici.

Esempio classe derivate:

```
class persona {
public:
    char nome [20];
    int eta;
};

// classe derivata studente, classe base persona
class studente : public persona{
public:
    int esami;
    int matricola;
};
```

BASE	char nome[20]	Anna	
	int eta	22	
DERIVATA	int esami	3	
	int matricola	7777	

oggetto di tipo studente

Figura 6.1: Un oggetto di una classe derivata ha tutti i campi della classe base più quelli della classe derivata.

Esempio di classi derivate di classi derivate:

```
class borsista : public studente{
public:
    int borsa;
    int durata;
};
```

BASE	char nome [20]	Anna	
	int eta	22	
DERIVATA	int esami	3	
	int matricola	7777	
	int borsa	500	
	int durata	3	

borsista

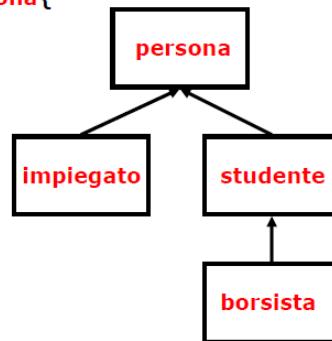
Istruzioni possibili

```
...
borsista b; borsista *pb;
b.borsa= 500;
pb->esami=33;
b.eta=22;
```

altro esempio

```
// classe derivata impiegato, classe base persona
```

```
class impiegato : public persona{
public:
    int livello;
    int stipendio;
};
```



Definizione 17 compatibilità fra tipi (puntatori): un oggetto (puntatore ad oggetto) di un tipo può essere convertito in un supertipo (puntaore ad un supertipo), ma non vale il viceversa

```
void main(){
    persona p;
    studente s;
    impiegato i;
    borsista b;
    p=s; // corretto : conversione implicita
          // da studente a persona
// s=p; errato : supertipo assegnato a sottotipo
// s=i; errato : tipi diversi
    p=b; // corretto : conversione implicita
          // da borsista a persona
    s=b; // corretto : conversione implicita
          // da borsista a studente
}
```

nome	Anna
eta	22
esami	3
matricola	7777

s

p=s;

nome	Anna
eta	22

p

Figura 6.2: Nella conversione i campi della classe derivata scompaiono

```
void main(){
    studente s; persona p; borsista b;
    studente* ps; persona * pp;
    pp=&p;
    ps =&s // corretto
    pp=ps; // corretto (conversione implicita)
    pp=&b; // corretto (conversione implicita)
    pp=new studente; // corretto (conversione implicita)
// ps =&p; errato
}
```



ps	→	nome	Anna
pp	→	eta	22
		esami	3
		matricola	7777

s

Figura 6.3: Con i puntatori, a differenza della conversione tra tipi fondamentali, i campi non scompaiono ma non sono più accessibili

(la scelta del campo a cui si accede avviene a tempo di compilazione in base al tipo del puntatore)

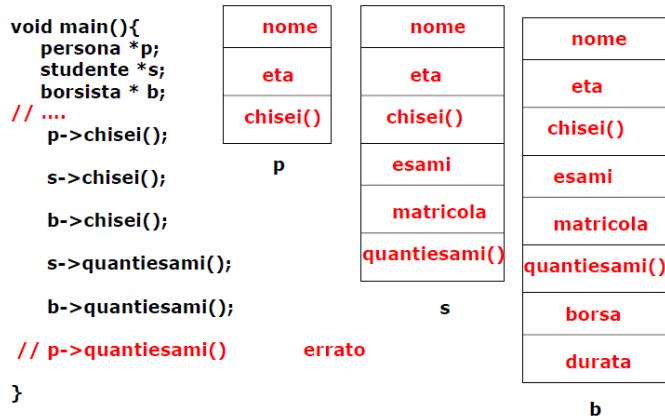


Figura 6.4: Classi derivate con funzioni membro

6.1 Regole di visibilità

```

class studente {
public:
    int matricola;
    int esami; // esami sostenuti
};

class borsista : public studente{
public:
    int borsa;
    int durata;
    int esami; // esami dall'inizio della borsa
};

void main(){
    studente * s=new studente;
    borsista * b=new borsista;
    b->esami=4; // = b.borsista::esami
    b->studente::esami=5; // risolutore di visibilità
    cout << b->esami; // 4
    s=b; // conversione
    cout << s->esami; // 5
}

```

più in generale...

```

class classe1 {
public:
    int a;
    //..
};

```

```

class classe2 : public classe1{
public:
    int a;
    // ...
};

class classe3 : public classe2{
public:
    int a;
    // ...
};

void main(){
    classe3 obj;
    obj.a=2; // obj.classe3::a
    obj.classe1::a=7;
    obj.classe2::a=8;
    cout << obj.a; // 2
    cout << obj.classe1::a; //7 risolutore di visibilità
    cout << obj.classe2::a; //8 risolutore di visibilità
}

```

6.1.1 Regole di visibilità con i puntatori

```

void main(){
    //...
    classe1* p1=&obj; // conversione
    classe2* p2=&obj; // conversione
    classe3* p3=&obj;
    cout << p1->a; // 7
    cout << p2->a; // 8
    cout << p3->a; // 2
}

```

6.1.2 Regole di visibilità con le funzioni membro

```

class persona {
public:
    char nome [20];
    int eta;
    void chisei(){
        cout << nome << '\t' << eta << endl;
    }
};

class studente : public persona{
public:
    int esami;
    int matricola;
}

```

```

void chisei(){
    cout << nome << '\t' << eta << '\t',
    << matricola <<
    '\t' << esami << endl;
}
;

void main(){
    studente s;
    strcpy(s.nome, "anna"); s.eta=22;
    s.esami=3; s.matricola=444444;
    s.chisei(); // anna 22 444444 3
                // chiamata a studente::chisei()
    s.persona::chisei(); // anna 22
    persona *p=&s;
    p->chisei(); // anna 22
}

```

altri esempi

```

#include<iostream.h>
class uno {
// ...
public:
    uno() { }
    void f(int) {
        cout << "uno";
    }
};
class due: public uno {
//...
public:
    due() {}
    void f() {
        cout << "due";
    }
};

void main (){
    due* p= new due;
//    p->f(6); errore
    p->uno::f(6); // uno
    p->f(); // due
}

```

6.2 specificatori di accesso

Definizione 18: I campi privati di una classe non sono accessibili dalle sottoclassi né dall'esterno

```

class uno {
    int x;
};
class due : public uno{
    int y;
    void f() {x=5; y=6; }//no perche' x e' privato di uno
};

```

Definizione 19: I campi protetti di una classe sono accessibili dalle sottoclassi, ma non dall'esterno

```

class uno {
protected:
    int x;
};
class due : public uno{
    int y;
    void f() {x=5; y=6; } // ok perche' x e' protetto
};
due * s= new due;
s->x=2 ; // no perche' x e' protetto ma non pubblico

```

Definizione 20: I campi pubblici di una classe sono accessibili dalle sottoclassi e anche dall'esterno

```

class uno {
public:
    int x;
};
class due : public uno{
public:
    int y;
    void f() {x=5; y=6; } // corretto perche' x e' pubblico
};
due * s= new due;
s->x=2 ; // corretto perche' x e' pubblico

```

I campi mantengono la stessa specifica in tutta la gerarchia.

6.3 costruzione degli oggetti

Quando un oggetto di una classe derivata viene costruito si costruisce prima la parte base e poi la parte derivata.

Costruzione di O:

- se O deriva da una classe base B: COSTRUZIONE (B);
- si costruiscono i campi di O chiamando gli opportuni costruttori nel caso che siano oggetti;

- si chiama il costruttore di O;

Se la classe base ha più costruttori, il costruttore di una classe derivata deve chiamarne uno nella lista di inizializzazione.

Può non chiamarlo esplicitamente se la classe base ha un costruttore di default, che in questo caso viene chiamato automaticamente.

```
class uno {
public:
    uno(){cout << "nuovo_>uno" << endl;}
};

class due: public uno {
public:
    due() {cout << "nuovo_>due" << endl;}
};

class tre: public due {
public:
    tre() {cout << "nuovo_>tre" << endl;}
};

void main (){
    due obj2; // nuovo uno
               // nuovo due
    tre obj3; // nuovo uno
               // nuovo due
               // nuovo tre
}
```

```
class uno {
protected:
    int a;
public:
    uno() {a=5; cout << "nuovo_>uno" << a << endl;}
    uno(int x) {a=x; cout << "nuovo_>uno" << a << endl;}
};

class due: public uno {
    int b;
public:
    due(int x) {b=x; cout << "nuovo_>due" << x << endl;}
};

void main (){
    due obj2(8); // nuovo uno 5
                  // nuovo due 8
}
```

costruzione di obj2

due obj2(8);

chiamata a uno::uno()

a	5
---	---

chiamata a due::due(8)

a	5
b	8

```
class uno {
protected:
    int a;
public:
    uno() {a=5; cout << "nuovo uno" << a << endl;}
    uno(int x) {a=x; cout << "nuovo uno" << a << endl;}
};

class due: public uno {
    int b;
public:
    due(int x): uno(x+1) {b=x; cout << "nuovo due" << x << endl;}
};

void main (){
    due obj2(8); // nuovo uno 9
                  // nuovo due 8
}
```

due obj2(8);

chiamata a uno::uno(9)

a	9
---	---

chiamata a due::due(8)

a	9
b	8

```
class uno {
public:
    uno(int x) {cout << "nuovo uno" << endl;}
};


```

```

class due: public uno {
public:
    // due(int x) {...} ERRORE: manca il costruttore di default
    // nella classe uno
};

```

Ordine di chiamata dei costruttori per una gerarchia a due livelli:

1. costruttori degli oggetti membri della classe base
2. costruttore della classe base
3. costruttori degli oggetti membri della classe derivata
4. costruttore della classe derivata

Costruttori con membri oggetto

```

class uno {
public:
    uno() {
        cout << "nuovo_>uno" << endl;
    }
};

class due {
    uno a;
public:
    due() {
        cout << "nuovo_>due" << endl;
    }
};

class tre: public due {
    uno b;
public:
    tre() { cout << "nuovo_>tre" << endl; }
};

void main (){
tre obj;
}
nuovo uno // uno::uno() per a
nuovo due // due::due() per obj
nuovo uno // uno::uno() per b
nuovo tre // tre::tre() per obj

```

Definizione 21 distruzione: quando un oggetto di una classe derivata viene distrutto viene distrutta prima la parte derivata e poi la parte base.

Distruzione di O:

- i campi di O vengono distrutti
- viene chiamato il distruttore di O

- se O deriva da una classe base B: DISTRUZIONE (B)

```

class uno {
public:
    uno();
    ~uno();
};

uno::uno(){cout << "nuovo\u2022uno" << endl;}
uno::~uno(){cout << "via\u2022uno" << endl;}

class due: public uno {
public:
    due();
    ~due();
};

due::due(){cout << "nuovo\u2022due" << endl;}
due::~due(){cout << "via\u2022due" << endl;}


void main (){
    due obj2;
    // nuovo uno
    // nuovo due
    // via due
    // via uno
}

```

6.4 membri statici

```

class A {
public:
    static int quantiA;
    A(){
        cout << "A\u2022=\u2022"
        << ++quantiA << endl;}
};

int A::quantiA=0;
class B : public A{
public:
    static int quantiB;
    B(){
        cout << "B\u2022=\u2022"
        << ++quantiB << endl;}
};

int B::quantiB=0;

void main(){
    A p1;
    // A = 1
}

```

```
B s1;  
// A = 2  
// B = 1  
A p2;  
// A = 3  
B s2;  
// A = 4  
// B = 2  
}
```

Capitolo 7

Funzioni virtuali, classi astratte e Polimorfismo (Lezione 3 Ducange)

7.1 Funzioni virtuali

Con le funzioni "normali" la scelta della funzione avviene a tempo di compilazione in base al tipo del puntatore

```
void main () {
    studente* s= new studente (5,777777);
    borsista* b= new borsista(10,888888,500000);
    studente* b1= b;
    s->chisei();
    // sono uno studente
    b->chisei();
    // sono un borsista
    b1->chisei(); // studente::chisei();
    // sono uno studente
}
```

in una gerarchia di classi, il metodo (la funzione) da chiamare viene scelto dinamicamente a tempo di esecuzione

```
class studente {
//...
public:
//...
    void virtual chisei() { cout << "sono uno studente";}
};

class borsista : public studente{
//...
public:
//...
```

```

    void virtual chisei() { cout << "sono un borsista";}
}; // virtual puo' mancare

void main () {
    studente* s= new studente (5,777777);
    borsista* b= new borsista(10,888888,500000);
    studente* b1= b;
    s->chisei();
    // sono uno studente
    b->chisei();
    // sono un borsista
    b1->chisei();
    // sono un borsista
}

```

La scelta della funzione (con le funzioni virtuali) avviene a tempo di esecuzione in base al tipo dell'oggetto effettivamente putato.

Tuttavia le funzioni virtuali non hanno effetto se sono chiamate dall'oggetto:

```

void main () {
    studente s(5,777777);
    borsista b(10,888888,500000);
    studente b1= b;
    s.chisei();
    // sono uno studente
    b.chisei();
    // sono un borsista
    b1.chisei();
    // sono uno studente
}
//b1 ha un solo campo "chisei"

```

esempio di utilizzo:

```

class studente {
//...
public:
//...
    int qualematicola(){
        return matricola;
    }
void virtual chisei() { cout << "sono uno studente";}
};
class borsista : public studente{
//...
public:
//...
    void chisei() { cout << "sono un borsista";}
};
void stampa (studente* s){
    s->chisei();
}

```

```

        cout << "matricola=";
        cout << s->qualematricola() << endl;
    }

void main(){
    studente* s [2];
    s[0] = new studente(7,77777);
    s[1] = new borsista(10,888888,500000);
    for(int i=0; i< 2; i++) stampa(s[i]);
}
//sono uno studente matricola=77777
//sono un borsista matricola=888888

```

7.1.1 Funzioni virtuali nella gerarchia

Una funzione è virtuale in tutte le classi che si trovano sotto quella che la definisce come virtuale

```

class uno {
//...
public:
    uno() {}
    void f() {
        cout << 1 << endl; }
};

class due : public uno{
public:
    due () {}
    void virtual f() {
        cout << 2 << endl; }
};

class tre: public due {
public:
    tre () {}
    void f() {
        cout << 3 << endl; }
};

void main(){
    due* p2= new tre;
    p2->f(); // 3 tre::f()
    uno* p1= new tre;
    p1->f(); // 1 uno::f()
}

```

nell'esempio f è virtuale in due e tre ma non in uno

7.1.2 Distruttori virtuali

```

class uno {
public:
    uno() {};
    virtual ~uno() {cout << "via_>uno" << endl;}
};

class due: public uno {
public:
    due() {};
    ~due() {cout << "via_>due" << endl
};

void main (){
    uno* obj=new due;
    //...
    delete obj;
}
// via due ~due()
// via uno
//senza virtual :
// via uno ~uno()

```

7.2 Classi astratte e polimorfismo

Definizione 22 classe astratta: serve come classe base nelle derivazioni.

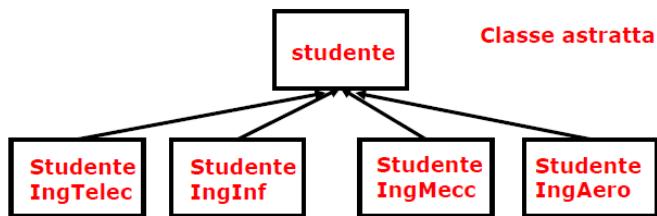
Viene specializzata nelle classi derivate.

Definisce una interfaccia unica verso le applicazioni.

Non viene definita completamente: ha almeno una funzione virtuale pura.

Non si possono istanziare oggetti di una classe astratta.

Definizione 23 funzione virtuale pura: è una funzione (ereditata o no) senza definizione: $F(\dots)=0$.



```

class studente {
    int matricola; int esami;
public:
    studente (int m){ esami=0; matricola=m; }
    // ...
    void virtual chisei() =0;
        // funzione virtuale pura

```

```

};

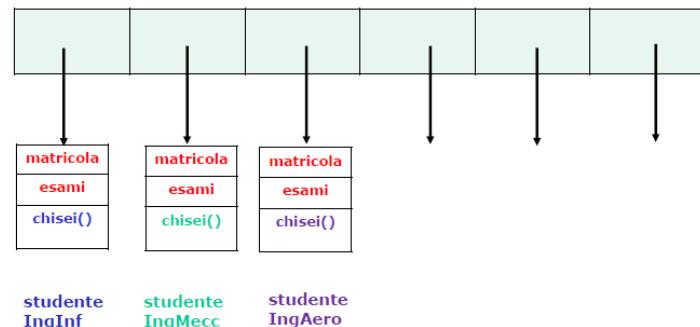
class studenteIngInf : public studente {
    //...
public:
    studenteIngInf(int m) : studente(m) {}
    // ...
    void chisei() {
        cout << "studente di ingegneria informatica" << endl;
    }
};

class studenteIngMecc : public studente{
    // ..
public:
    studenteIngMecc(int m) : studente(m) {}
    // ...
    void chisei() {
        cout << "studente di ingegneria meccanica" << endl;
    }
};

void main(){
// studente s; errato studente e' una classe astratta
studente* s; // OK viene dichiarato un puntatore
studente* studenti [3];
studenti[0]= new studenteIngInf(777777) ;
studenti[1]= new studenteIngMecc(888888);
studenti[2]= new studenteIngInf(888888) ;
for (int i=0; i<3; i++)
    studenti[i]->chisei();
}
//studente di ingegneria informatica
//studente di ingegneria meccanica
//studente di ingegneria informatica

```

array con elementi di tipo **studente***



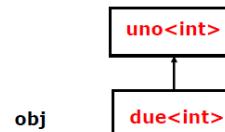
classi modello e derivazione: classe base modello, classe derivata modello con lo stesso tipo

```
template <class T>
class uno {
    T a;
public:
    uno(T x) {
        a=x;
        cout << a << endl;
    }
};

template <class tipo>
class due: public uno<tipo> {
    tipo b;
public:
    due(tipo x, tipo y);
    uno<tipo>(x) {
        b=y; cout << b << endl;
    }
};
```

```
void main (){
    due<int> obj(7,8);
}

7 uno<int>::uno<int>(7)
8 due<int>::due<int>(7,8)
```



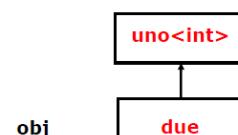
classi modello e derivazione: classe base istanziata, classe derivata non modello

```
template <class T>
class uno {
    T a;
public:
    uno(T x) {
        a=x;
        cout << a << endl;
    }
};

class due: public uno<int> {
    int b;
public:
    due(int x, int y);
    uno<int>(x) {
        b=y; cout << b << endl;
    }
};
```

```
void main (){
    due obj(7,8);
}

7 uno<int>::uno<int>(7)
8 due::due(7,8)
```



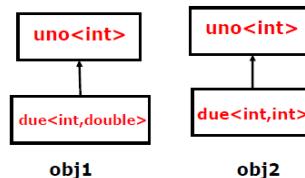
classi modello e derivazione: classe base modello, classe derivata modello

```
template <class T>
class uno {
    T a;
public:
    uno(T x) {
        a=x;
        cout << a << endl;
    }
};

template <class tipo1, class tipo2>
class due: public uno<tipo1> {
    tipo2 b;
public:
    due(tipo1 x, tipo2 y);
    uno<tipo1>(x) {
        b=y; cout << b << endl;
    }
};
```

```
void main (){
    due<int,double> obj1(7,8.5);
    due<int,int> obj2(7,8.5);
}

7 uno<int>::uno(7)
8.5 due<int,double>::due<int,double>(7,8.5)
7 uno<int>::uno(7)
8 due<int,int>::due<int,int>(7,8.5)
```



7.3 Gestione delle eccezioni

Definizione 24 Eccezioni: Errori a runtime (es. divisione per 0, indice array fuori dall'intervallo)

Situazioni anomale non rilevabili dal compilatore.

Possono causare il crash dell'applicazione.

Definizione 25 Eccezioni: costrutto sintattico: Possibilità di individuare le eccezioni e gestirle da programma a tempo di esecuzione.

Metodo formale e ben definito.

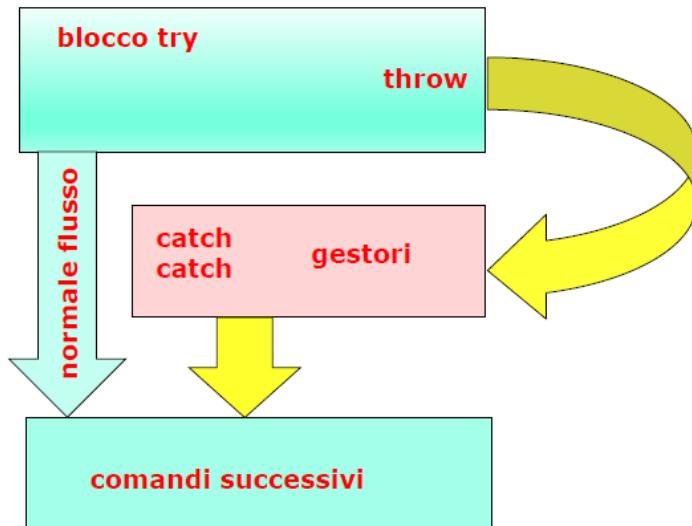
Netta separazione tra il codice che rileva l'eccezione e il codice che lo gestisce.

```
try {
    //...
    throw espressione1; // lancia eccezione
    ...
    throw espressionem; // lancia eccezione
    //...
}
catch (tipo1 e) { ... } // gestione eccezione
//...
catch (tipon e) { ... } // gestione eccezione
//comandi_successivi ...
```

throw= lancia un'eccezione

Eccezioni: costrutto sintattico:

- Se viene lanciata una eccezione (throw), l'esecuzione del blocco try si interrompe
- le eccezioni lanciate durante l'esecuzione del blocco try sono gestite dai gestori (clausole catch): la gestione è scelta in base al tipo dell'eccezione lanciata
- dopo la gestione dell'eccezione, l'esecuzione prosegue normalmente con comandi successivi non considerando le altre clausole catch
- se nessuna eccezione viene lanciata, l'esecuzione prosegue con comandi successivi
- se un'eccezione lanciata non viene catturata, il programma termina con errore
- Un'eccezione può essere lanciata soltanto durante l'esecuzione di un blocco try



```

void div (int x, int y){
    try {
        if (y==0) throw "divisione per 0"; // è una stringa
        // che rappresenta il tipo dell'eccezione (const char * p)
        cout << x/y << endl;
    }
    catch (const char* p) { cout << p << endl; }
    cout << "fine div" << endl ;
}
void main(){
    int x,y;
    cin >> x >> y;
    div(x,y);
    cout << "fine main";
}

```

con 10 5: 2, fine div, fine main.

con 10 0: divisione per 0, fine div, fine main.

Esempio: divisione per 0 (II): eccezione non catturata

```
void div (int x, int y){  
    try {  
        if (y==0) throw 0; // nota: eccezione intera  
        cout << x/y << endl;  
    }  
    catch (const char* p) { cout << p << endl; }  
  
    cout << "fine div" << endl ;  
}  
  
void main(){  
    int x,y;  
    cin >> x >> y;  
    div(x,y);  
    cout << endl << "fine main";  
}  
  
con 10 0 il programma termina con errore
```

Esempio: divisione per 0 (III)

```
void div (int x, int y){  
    try {  
        if (y==0) throw 0;  
        cout << x/y << endl;  
    }  
  
    // nota  
    catch (int) {  
        cout << "divisione per 0" << endl; }  
  
    cout << "fine div" << endl ;  
}  
  
void main(){  
    int x,y;  
    cin >> x >> y;  
    div(x,y);  
    cout << endl << "fine main";  
}
```

con 10 0:
divisione per 0
fine div
fine main

Esempio: divisione per 0 o negativa (IV)

```
void positive_div (int x, int y){  
    try {  
        if (y==0) throw 0;  
        if ( (x<0 && y>0) || (x>0 && y<0) ) throw 1;  
        cout << x/y << endl;  
    }  
    catch (int n) {  
        if (n==0) cout << "divisione per 0" << endl;  
        else cout << "risultato negativo" << endl;  
    }  
    cout << "fine div" << endl ;  
}  
  
void main(){  
    int x,y;  
    cin >> x >> y;  
    positive_div(x,y);  
    cout << "fine main";  
}
```

con input -2 3
risultato negativo
fine div
fine main

Esempio: divisione per 0 (V)

```

void positive_div (int x, int y){
    try {
        if (y==0) throw 0; // intero
        if ( (x<0 && y>0) || (x>0 && y<0) ) throw '0'; // carattere
        cout << x/y << endl;
    }
    catch (int) {
        cout << "divisione per 0" << endl;
    }
    catch (char) {
        cout << "risultato negativo" << endl;
    }
    cout << "fine div" << endl;
}
void main(){
    int x,y;
    cin >> x >> y;
    positive_div(x,y);
    cout << "fine main";
}

```

con input -2 3

risultato negativo
fine div
fine main

Throw w try-catch in funzioni diverse

```

void div (int x, int y){
    // lancio eccezione
    if (y==0) throw "divisione per 0";
    cout << x/y << endl;
    cout << "fine div" << endl ;
}

void main(){
    try { // gestione eccezione nel main
        int x,y;
        cin >> x;
        cin >> y;
        div(x,y);
    }
    catch (const char* p) {
        cout << p << endl;
    }
    cout << endl << "fine main";
}

```

con 10 0: divisione per 0, fine main.

Corrispondeza fra throw e catch:

- no conversioni implicite (a parte sottotipo -> sopratipo)
- L'eccezione viene gestita a tempo di esecuzione esaminando i gestori nell'ordine in cui compaiono a partire dal blocco più recente incontrato.
- Viene scelto il primo con argomento corrispondente all'eccezione lanciata

Ordine dei gestori

```

void f(int x) {
    if (x==0) throw x;
    if (x>100) throw 'a';
    cout << "fine f" << endl;
}
void g(int x) {
    try { f(x);}
    catch(int) {
        cout << "eccezione da g"
        << endl; }
    cout << "fine g" << endl;
}
void main(){
    try { int x; cin >> x; g(x);}
    catch(char) {
        cout << "eccezione da main"
        << endl; }
    cout << "fine main";
}

```

con input 0:
eccezione da g
fine g
fine main

con input 200:
eccezione da main
fine main

Ordine dei gestori

```

void f(int x) {
    if (x==0) throw x;
    if (x>100) throw 'a';
    cout << "fine f" << endl;
}
void g(int x) {
    try { f(x);}
    catch(int) {
        cout << "eccezione da g"
        << endl; }
    cout << "fine g" << endl;
}
void main(){
    try { int x; cin >> x; g(x);}
    catch(int) {
        cout << "eccezione da main"
        << endl; }
    cout << "fine main";
}

```

con input 0:
eccezione da g
fine g
fine main

con input 200:
errore



Stack di esecuzione

Clausola catch generica: cattura qualsiasi eccezione:

```
void main(){
    try {
        int x; cin >> x;
        if (x==0) throw x;
        if (x <0) throw 7.8;
    }
    catch(int) {
        cout << "eccezione da main" << endl;
    }
    catch(...) {
        cout << "eccezione non prevista da main" << endl;
        cout << "fine main";
    }
//con input=-1:
//eccezione non prevista da main fine main
```

Capitolo 8

Programmi ricorsivi (lezione 5 virdis)

esempi base fattoriale di n:

definizione iterativa = $1 \times 2 \times \dots \times n$ se $n > 0$

definizione induttiva o ricorsiva: $n! = (n * (n-1)!)$ se $n > 0$.

Definizione 26 Regole sulla ricorsività: (vanno rispettate tutte e 3)

1. individuare i casi base in cui la funzione è definita immediatamente
2. effettuare le chiamate ricorsive su un insieme più "piccolo" di dati
3. fare in modo che alla fine di ogni sequenza di chiamate ricorsive si ricada in uno dei casi base

```
//esempi di errori
//es 1: non e' rispettata la 3

int pari_errata(int x) {
    if (x == 0) return 1;
    return pari_errata(x-2);
}

//es 2 non e' rispettata la 2
int MCD_errata(int x, int y) {
    if (x == y) return x;
    if (x < y) return MCD_errata(x, y-x);
    return MCD_errata(x, y);
}
```

Definizione 27 lista: .

- NULL (sequenza vuota) è una lista

- un elemento seguito da una lista è una lista

-> le liste sono oggetti che possono essere rappresentati in maniera ricorsiva
(esempi vari programmi semplici sulle liste ricorsivi pdf)

Induzione naturale

Sia P una proprietà sui naturali.

Base. P vale per 0

Passo induttivo. per ogni naturale n è vero che:

Se P vale per n allora P vale per (n+1)



P vale per tutti i naturali

Definizione 28 Complessità dei programmi ricorsivi: $T(0) = a$
 $T(n) = b + T(n-1)$

esempio:

$T(0) = a$ $T(1) = b + a$ $T(2) = b + b + a = 2b + a$ $T(3) = b + 2b + a = 3b + a \dots T(n) = nb + a$
-> $T(n) \in O(n)$

8.1 QuickSort

vedi:67

8.2 Ricerca in un insieme

Definizione 29 Ricerca lineare ricorsiva: scorre l'array.
Complessità = $O(n)$

```
int RlinearSearch (int A [], int x, int m, int i=0) {
    if (i == m) return 0;
    if (A[i] == x) return 1;
    return RlinearSearch(A, x, m, i+1);
}
```

Definizione 30 ricerca binaria ricorsiva: Complessità = $O(\log n)$

```
int binSearch (int A [],int x, int i=0, int j=m-1)
{
    if (i > j) return 0;
    int k=(i+j)/2;
    if (x == A[k]) return 1;
    if (x < A[k])
        return binSearch(A, x, i, k-1);
    else
        return binSearch(A, x, k+1, j);
}
```

Capitolo 9

MergeSort, ordinamento STL e gestione liste (Lezione 6 Virdis)

9.1 Merge Sort

68

9.1.1 nozioni aggiuntive

- con l'insertion sort (mano occhio) dal punto di vista asintotico non cambia nulla , tuttavia lavorare su array piú piccoli é vantaggioso (il limite infatti é superiore -> su numeri piú piccoli possiamo trarre dei vantaggi)
- considerare due strutture dati separate (i due o più sottoarray) consente di parallelizzare il calcolo
- **complessità:**
 - elementi = n
 - livelli = $\log n + 1$
 - costo livello n

$$n(\log(n) + 1) \rightarrow n\log(n) + n$$

	Worst Case	Best Case	Average Case
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$

9.2 Complessità

- Tempo di esecuzione: worst vs best vs avg
- Memoria: in-place (=non alloca memoria aggiuntiva) or not in-place (=alloca memoria aggiuntiva per lavorare)

9.3 Compiler Flags

- g++ -W -o test test.cpp
- g++ -Wall -W -o test test.cpp

sono 2 flag che permettono di dare dei warning sull'analisi statica del codice

9.4 STL sort

sort (first, last);
overloading: sort (first, last, comparatore);
(comparatore è una funzione booleana)

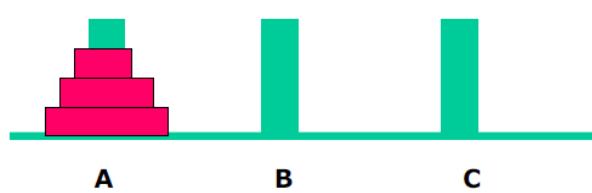
9.5 Valgrind

Capitolo 10

Torre di Hanoi, ricerca in un insieme e metodo "dividi et impera"

10.1 torre di Hanoi (Lezione 7 Virdis)

- 3 paletti, 1 torre di n cerchi
- spostare la torre dal paletto sorgente A a quello destinatario C usando un paletto ausiliario B
- Un cerchio alla volta
- Mai un cerchio sopra uno più piccolo



```

void trasferisci una torre di n cerchi da A a C
{
    Se n=1                                         caso base
        sposta il cerchio dal A a C;

    altrimenti
    {
        trasferisci la torre degli n-1 cerchi più piccoli da A a B
        usando C come paletto ausiliario;
        sposta il cerchio più grande dal A a C;
        trasferisci la torre degli n-1 cerchi più piccoli da B a C
        usando A come paletto ausiliario;
    }
}

```

Figura 10.1: Pseudo-codice

```

void hanoi(int n, pal A, pal B, pal C)
{
    if (n == 1)
        sposta(A, C);
    else {
        hanoi(n - 1, A, C, B);
        sposta(A, C);
        hanoi(n - 1, B, A, C);
    }
}

```

Complessità:

$$T(1) = a$$

$$T(n) = b + 2T(n-1)$$

Quindi:

$$T(1) = a$$

$$T(2) = b + 2a$$

$$T(3) = b + 2b + 4a = 3b + 4a$$

$$T(4) = 7b + 8a$$

.

$$T(n) = (2^{(n-1)} - 1)b + 2^{(n-1)}a$$

$$T(n) \in O(2^n)$$

10.2 ricerca in un insieme

Ricerca lineare

```

int RlinearSearch (int A [], int x, int m, int i=0) {

```

```

    if (i == m) return 0;
    if (A[i] == x) return 1;
    return RlinearSearch(A, x, m, i+1);
}

```

$T(0) = a$
 $T(n) = b + T(n-1)$
 Quindi è $O(n)$
ricerca binaria

```

int binSearch (int A [], int x, int i=0, int j=m-1)
{
    if (i > j) return 0;
    int k=(i+j)/2;
    if (x == A[k]) return 1;
    if (x < A[k])
        return binSearch(A, x, i, k-1);
    else
        return binSearch(A, x, k+1, j);
}

```

$T(0) = a$
 $T(n) = b + T(n/2)$
 Quindi $T(n)$ è $O(\log n)$
ricerca

```

int Search (int A [], int x, int i=0, int j=n-1) {
    if (i > j) return 0;
    int k=(i+j)/2;
    if (x == A[k])
        return 1;
    return Search(A, x, i, k-1) || Search(A, x, k+1, j);
}

```

$T(0) = a$
 $T(n) = b + 2T(n/2)$
 Quindi $T(n)$ è $O(n)$

10.3 Dividi et impera

```

void dividetimpera( S )
{
    if ( |S| <= m )
        <risolvi direttamente il problema>;
    else {
        <dividi S in b sottoinsiemi S1.. Sb>;
        dividetimpera(S i1 );
        ...
        dividetimpera(S ia );
    }
}

```

```

        } <combina i risultati ottenuti>;
}

```

10.3.1 Classificazione di alcune relazioni di ricorrenza (non lineari)

$$T(0) = d \quad O(\log n)$$

$$T(n) = c + T(n/2)$$

$$T(0) = d$$

$$T(n) = c + 2T(n/2) \quad O(n)$$

$$T(0) = d$$

$$T(n) = cn + 2T(n/2) \quad O(n \log n)$$

$T(n) = d$	se $n = 1$
$T(n) = c + aT(n/b)$	se $n > 1$

$$T(n) \in O(\log n) \quad \text{se } a = 1$$

$$T(n) \in O(n^{\log_b a}) \quad \text{se } a > 1$$

$T(n) = d$	se $n \leq m$
$T(n) = hn^k + aT(n/b)$	se $n > m$

$h > 0$

$$T(n) \in O(n^k) \quad \text{se } a < b^k$$

$$T(n) \in O(n^k \log n) \quad \text{se } a = b^k$$

$$T(n) \in O(n^{\log_b a}) \quad \text{se } a > b^k$$

10.3.2 Classificazione di alcune relazioni di ricorrenza (lineari)

$$\begin{aligned} T(0) &= d \\ T(n) &= b + T(n-1) \end{aligned} \quad \text{O}(n)$$

$$\begin{aligned} T(1) &= a \\ T(n) &= bn + T(n-1) \end{aligned} \quad \text{O}(n^2)$$

$$\begin{aligned} T(0) &= d \\ T(n) &= b + 2T(n-1) \end{aligned} \quad \text{O}(2^n)$$

$$\begin{aligned} T(0) &= d \\ T(n) &= bn^k + a_1T(n-1) + a_2T(n-2) + \dots + a_rT(n-r) \end{aligned}$$

Polinomiale solo se

- esiste al più un solo $a_i = 1$ e
- gli altri a_i sono tutti 0 (c'è una sola chiamata ricorsiva).

Negli altri casi sempre **esponenziale**.

$$T(0) = d$$

$$T(n) = bn^k + T(n-1)$$

$$b > 0$$

$$T(n) \in O(n^{k+1})$$

10.4 Algoritmi di teoria dei numeri

La complessità è calcolata prendendo come misura il numero di cifre che compongono il numero; ad esempio:

- L'addizione ha complessità $O(n)$
- la moltiplicazione che studiamo alle elementari ha complessità $O(n^2)$

(esempi pag 21 pdf)

Capitolo 11

Serie di Fibonacci e Merge sort (Lezione 8 Virdis)

(vedi pdf lezione abbastanza useless)

Capitolo 12

Alberi binari (Lezione 4 Ducange)

12.1 Intro

Definizione 31 Albero binario: .

- NULL è un albero binario
 - un nodo p più due alberi binari Bs e Bd forma un albero binario
- si dice binario in quanto da un nodo al più si ha un binary splitting

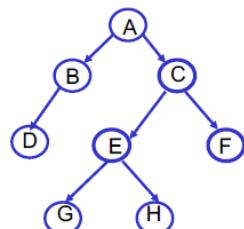
p è radice

Bs è il sottoalbero sinistro di p

Bd è il sottoalbero destro di p

Elementi di un albero

- padre = puntatore al nodo iniziale
- figlio sinistro (e figlio destro)
- antecedente
- foglia = nodo che non ha figli
- discendente
- livello di un nodo = corrisponde al numero dei suoi antecedenti
- livello dell'albero = massimo dei livelli dei suoi nodi



Altre nozioni:

- si assume che il livello di un albero vuoto è -1
- il livello della radice è 0
- il livello dell'albero è il più lungo cammino fra la radice e una foglia
- un albero binario etichettato è un albero binario in cui ad ogni nodo è associato un nome o etichetta

12.2 Ricorsione su alberi binari

Caso base = albero vuoto

Caso ricorsivo = radice + due sottoalberi

12.3 Visite di alberi binari

Le operazioni più comuni sugli alberi sono quelle di linearizzazione (= restituire la sequenza dei valori contenuti all'interno dell'albero), ricerca, inserimento e cancellazione di nodi.

una linearizzazione di un albero è una sequenza contenente i nomi dei suoi nodi.

Le più comuni linearizzazioni, dette visite, degli alberi binari sono tre:

- ordine anticipato (preorder)
- ordine differito (postorder)
- ordine simmetrico (inorder)

12.3.1 Memorizzazione in lista multipla

```
struct Node {
    InfoType label;
    Node* left;
    Node* right;
};
```

12.3.2 Visita anticipata (preorder)

(utilizziamo come convenzione una visita che parte prima da sx e va verso dx)

```
void preOrder(Node* tree){
    if (!tree) return;
    else {
        <esamina tree->label>; //cout<<tree->label;
        preOrder(tree->left);
```

```

        preOrder(tree->right);
    }
}
//A B D C E G H F

```

12.3.3 Visita differita (postorder)

```

void postOrder(Node* tree) {
    if (!tree) return;
    else {
        postOrder(tree->left);
        postOrder(tree->right);
        <esamina tree->label>;
    }
}
//D B G H E F C A

```

12.3.4 Visita simmetrica (inorder)

```

void inOrder(Node* tree) {
    if (!tree) return;
    else {
        inOrder(tree->left);
        <esamina tree->label>;
        inOrder(tree-> right);
    }
}
//D B A G E H C F

```

12.3.5 Complessità delle visite

Complessità in funzione del numero di nodi:

$$T(0) = a$$

$$T(n) = b + T(n_s) + T(n_d) \quad \text{con } n_s + n_d = n-1 \quad n > 0$$

Caso particolare:

$$T(0) = a$$

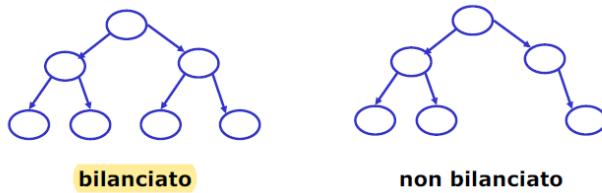
$$T(n) = b + 2T((n-1)/2)$$

$T(n) \in O(n)$

12.4 Bilanciamento degli alberi

Definizione 32 Albero binario bilanciato: i nodi di tutti i livelli tranne quelli dell'ultimo hanno due figli

(def appunti: alberi in cui i nodi di tutti i livelli (tranne le foglie) hanno 2 figli e hanno lo stesso livello)



Un albero binario bilanciato con livello k ha $2^{k+1} - 1$ nodi e 2^k foglie

Definizione 33 Alberi binari quasi bilanciati: fino al penultimo livello è un albero bilanciato (un albero bilanciato è anche quasi bilanciato)

Definizione 34 Alberi pienamente binari: tutti i nodi tranne le foglie hanno 2 figli

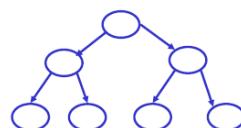
(un albero binario pienamente binario ha tanti nodi interni quante sono le foglie meno 1)

12.5 Complessità delle visite nel numero dei livelli

Complessità in funzione dei livelli (se l'albero è bilanciato):

$$T(0) = a$$

$$T(k) = b + 2T(k-1)$$



$$T(k) \in O(2^k)$$

12.6 Funzioni su alberi

12.6.1 conta i nodi

```
int nodes (Node* tree) {  
    if (!tree) return 0; // albero vuoto
```

```

        return 1+nodes(tree->left)+nodes(tree->right);
}

```

12.6.2 conta le foglie

```

int leaves (Node* tree) {
    if (!tree) return 0; // albero vuoto
    if ( !tree->left && !tree->right ) return 1; // foglia
    return leaves(tree->left)+leaves(tree->right);
}

```

Queste due funzioni hanno complessità O(n)

12.6.3 Cerca un'etichetta

Restituisce il puntatore al nodo che contiene l'etichetta n. Se l'etichetta non compare nell'albero restituisce NULL. SE più nodi contengono n, restituisce il primo nodo che si incontra sta facendo la visita anticipata.

```

Node* findNode (Infotype n, Node*tree) {
    if (!tree) return NULL; //albero vuoto: l' etichetta non c'e'
    if (tree->label==n) // trovata: restituisce il puntatore
        return tree;
    Node* a=findNode(n, tree->left); // cerca a sinistra
    if (a) return a; // se trovata restituisce il puntatore
    else return findNode(n, tree->right); // cerca a destra
}

```

12.6.4 Cancella tutto l'albero

Visita in postorder

alla fine il puntatore deve essere NULL

```

void delTree(Node* &tree) {
    if (tree) {
        delTree(tree->left);
        delTree(tree->right);
        delete tree;
        tree=NULL;
    }
}

```

12.6.5 Inserisci un nodo

Inserisce un nodo (son) come il figlio di father, sinistro se c = 'l', destro se c = 'r'. Restituisce 1 se l'operazione ha successo, 0 altrimenti. Se l'albero è vuoto inserisce il nodo come radice. Se father non compare nell'albero o ha già un figlio in quella posizione non modifica l'albero

```

int insertNode (Node* & tree, InfoType son,
InfoType father, char c){
    if (!tree) { // albero vuoto
        tree=new Node;
        tree->label=son;
        tree->left = tree->right = NULL;
        return 1;
    }
    Node* a=findNode(father,tree); //cerca father
    if (!a) return 0; //father non c'e'
    if (c=='l' && !a->left) { //inserisci come figlio sinistro e verifica
        che non esista gia un figlio
        a->left=new Node;
        a->left->label=son;
        a->left->left = a->left->right=NULL; //imposta la foglia
        return 1;
    }
    if (c=='r' && !a->right) { //inserisci come figlio destro
        a->right=new Node;
        a->right->label=son;
        a->right->left = a->right->right = NULL;
        return 1;
    }
    return 0; //inserimento impossibile
}

```

12.7 Classe Bin Tree

```

template<class InfoType>
class BinTree {
    struct Node {
        InfoType label;
        Node *left, *right;
    };
    Node *root;
    Node* findNode(InfoType, Node*);
    void preOrder(Node*);
    void inOrder(Node*);
    void postOrder(Node*);
    void delTree(Node*&);

    int insertNode(Node*&, InfoType, InfoType, char)

```

```
public:  
    BinTree() { root = NULL; }  
    ~BinTree(){ delTree(root); }  
    int find(InfoType x) { return findNode(x, root); }  
    void pre() { preOrder(root); }  
    void post(){ postOrder(root); }  
    void in() { inOrder(root); }  
    int insert(InfoType son, InfoType father, char c) {  
        insertNode(root,son,father,c);  
    };  
};
```

Capitolo 13

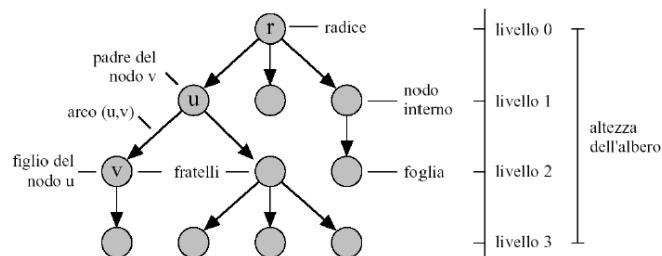
Alberi generici (Lezione 5 Ducange)

13.1 intro

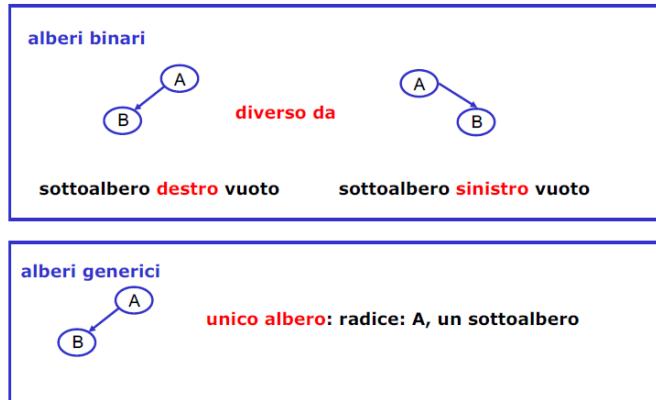
un albero binario o è un insieme di nodi vuoto o è un insieme di nodi; esistono anche degli alberi non binari: formalmente la differenza è: "un insieme di nodi NON vuoto" (inoltre ogni padre può avere più di due figli -> tutti i nodi con lo stesso padre si chiamano fratelli)

Definizione 35 Albero generico:

- un nodo p è un albero
- un nodo + una sequenza di alberi A1...An è un albero



13.1.1 Differenza con alberi binari



13.1.2 Visite

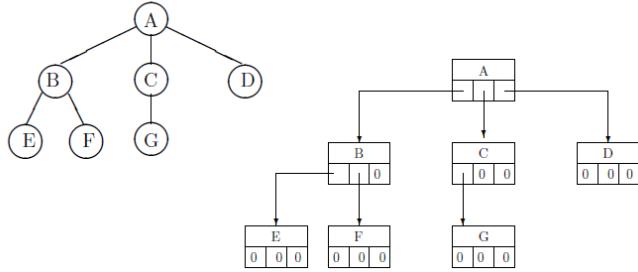
```
void preOrder ( albero ) {  
    esamina la radice;  
    se l'albero ha n sottoalberi {  
        preOrder ( primo sottoalbero );  
        ...  
        preOrder ( n-esimo sottoalbero );  
    }  
}
```

```
void postOrder ( albero ) {  
    se l'albero ha n sottoalberi {  
        postOrder ( primo sottoalbero );  
        ...  
        postOrder ( n-esimo sottoalbero );  
    }  
    esamina la radice;  
}
```

13.1.3 memorizzazione

Memorizzazione a liste multiple: problema di spreco di memoria con tanti puntatori a null inutili

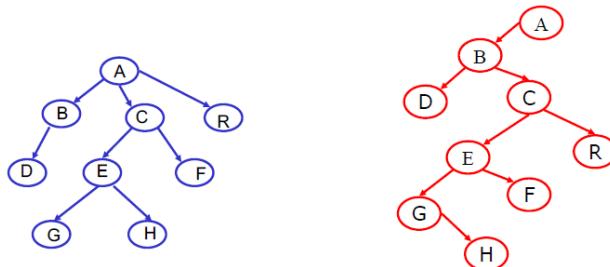
MEMORIZZAZIONE a liste multiple



Memorizzazione figlio-fratello:

- primo figlio a sinistra
- primo fratello a destra

risoluzione del problema della memoria: memorizzo l'albero generico in un albero binario



13.1.4 corrispondenza tra visite

(utilizzando la memorizzazione figlio-fratello)

La visita preorder del trasformato corrisponde alla visita preorder dell'albero generico.

La visita inorder del trasformato corrisponde alla visita postorder dell'albero generico.

Il tempo delle visite in un albero generico è lineare nel numero dei nodi.

Per la ricerca, l'inserimento e la cancellazione di un nodo, il tempo è comunque lineare. Infatti queste operazioni possono essere programmate mantenendo la struttura delle visite.

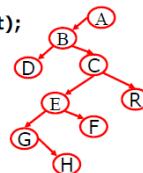
13.1.5 esempi

conta i nodi (vedi albero binario)

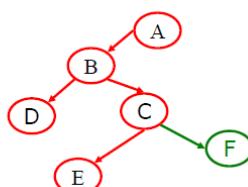
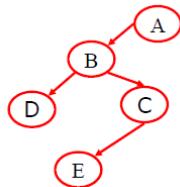
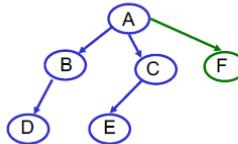
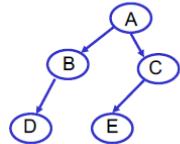
```
int nodes (Node* tree) {
    if (!tree) return 0;
    return 1+nodes(tree->left)+nodes(tree->right);
}
```

conta le foglie

```
int leaves(Node* tree) {
    if (!tree) return 0;
    if (!tree->left) return 1+ leaves(tree->right); // foglia
    return leaves(tree->left)+ leaves(tree->right);
}
```

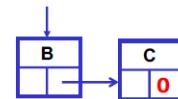


Inserisci F come ultimo figlio di A



inserisce un nodo in fondo a una lista di fratelli

```
void addSon(InfoType x, Node* &list) {
    if (!list) { //lista vuota
        list=new Node;
        list->label=x;
        list->left = list->right = NULL;
    }
    else //lista non vuota
        addSon(x, list->right);
}
```



inserisce son come ultimo figlio di father.

```
int insert(InfoType son, InfoType father, Node* &tree) {
    Node* a=findNode(father, tree); // a: puntatore di father
    if (!a) return 0; // father non trovato
    addSon(son, a->left);
    return 1;
}
```

13.2 Alberi binari di ricerca

Definizione 36 Albero binario di ricerca: è un albero binario tale che per ogni nodo p:

- i nodi del sottoalbero sinistro di p hanno etichetta minore dell'etichetta di p
- i nodi del sottoalbero destro di p hanno etichetta maggiore dell'etichetta di p

Dalla proprietà base segue che i nodi di un albero binario di ricerca hanno tutti etichette diverse (nei casi in cui la definizione degli alberi di ricerca sia con il \geq o \leq ci possono essere dei duplicati)

13.2.1 Proprietà e operazioni

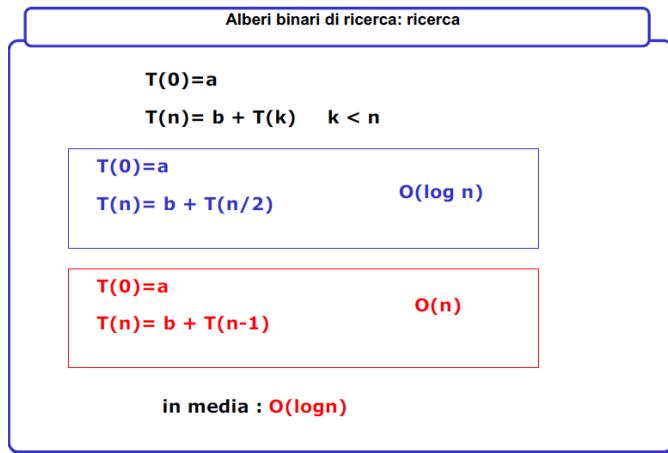
Definizione 37 proprietà: .

- non ci sono doppioni
- la visita simmetrica elenca le etichette in ordine crescente

Definizione 38 Operazioni: .

- ricerca di un nodo
- inserimento di un nodo
- cancellazione di un nodo

```
//ricerca
Node* findNode (InfoType n, Node* tree) {
    if (!tree) return 0; // albero vuoto
    if (n == tree->label) return tree; // n=radice
    if (n<tree->label) // n<radice
        return findNode(n, tree->left);
    return findNode(n, tree->right); // n>radice
}
```



```

void insertNode (InfoType n, Node* &tree) {
    if (!tree) { // albero vuoto: creazione nodo
        tree=new Node;
        tree->label=n;
        tree->left = tree->right = NULL; return;
    }
    if (n<tree->label) // n<radice
        insertNode (n, tree->left);
    if (n>tree->label) // n>radice
        insertNode (n, tree->right);
}
//complessita'  $O(\log n)$ 

```

Cancellazione:

- Prima si cerca il nodo da cancellare effettuando una ricerca come negli algoritmi precedenti.
- Se il nodo viene trovato, sia esso p, possono verificarsi due situazioni diverse.
- Se p ha un sottoalbero vuoto, il padre di p viene connesso all'unico sottoalbero non vuoto di p
- Se p ha entrambi i sottoalberi non vuoti si cerca il nodo con etichetta minore nel sottoalbero destro di p, si cancella e si mette la sua etichetta come etichetta di p

```

//restituisce l'etichetta del nodo piu' piccolo di un albero ed
//elimina il nodo che la contiene
void deleteMin (Node* &tree, InfoType &m) {
    if (tree->left) //c'e' un nodo piu' piccolo
        deleteMin(tree->left, m);
    else {

```

```

        m=tree->label; //restitusco l etichetta
        Node* a=tree;
        tree=tree->right; //connetto il sottoalbero destro di
        // m al padre di m
        delete a; //elimino il nodo
    }
}

void deleteNode(InfoType n, Node* &tree) {
    if (tree)
        if (n < tree->label) //n minore della radice
            { deleteNode(n, tree->left); return; }
        if (n > tree->label) //n maggiore della radice
            { deleteNode(n, tree->right); return; }
        if (!tree->left) //n non ha figlio sinistro
            { Node* a=tree; tree=tree->right; delete a;return }
        if (!tree->right) //n non ha figlio destro
            { Node* a=tree; tree=tree->left; delete a; return }
        deleteMin (tree->right, tree->label); //n ha entrambi i figli
    }
//O(logn)
}

```

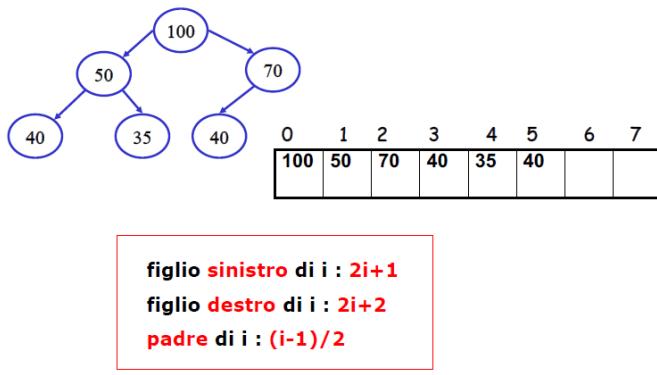
Capitolo 14

heap (Lezione 6 Ducange)

Definizione 39 heap: albero binario quasi bilanciato con le seguenti proprietà:

- i nodi dell'ultimo livello sono addossati a sinistra
- in ogni sottoalbero l'etichetta della radice è maggiore o uguale a quella di tutti i discendenti

Caratteristica importante dello heap: può essere facilmente rappresentato (linearizzato) in un vettore



14.1 Classe Heap e operazioni

Operazioni:

- inserimento di un nodo
- estrazione dell'elemento maggiore (radice)

Classe heap:

```

class Heap {
    int * h;
    int last; // indice dell'ultimo elemento
    void up(int);
    void down(int);
    void exchange(int i, int j){
        int k=h[i]; h[i]=h[j];h[j]=k;
    }
public:
    Heap(int);
    ~Heap();
    void insert(int);
    int extract();
};

```

Costruttore e distruttore:

```

Heap::Heap(int n){
    h=new int[n];
    last=-1;
}

Heap::~Heap() {
    delete [] h;
}

```

14.1.1 Inserimento

- memorizza l'elemento nella prima posizione libera dell'array
- fa risalire l'elemento tramite scambi figlio-padre per mantenere le proprietà dello heap

```

void Heap::insert (int x) {
    h[++last]=x;
    up(last);
}
// in questo codice manca solo il controllo del vettore pieno

```

Funzione up (riordina padri e figli dopo l'inserimento):

```

void Heap::up(int i) { // i e' l'indice dell'elemento da far risalire
    if (i > 0) // se non sono sulla radice
        if (h[i] > h[(i-1)/ 2]) { // se l'elemento e' maggiore del padre
            exchange(i,(i-1)/2); // scambia il figlio col padre
            up((i-1)/2); // e chiama up sulla nuova posizione
        } // altrimenti termina
}

```

- la funzione termina o quando viene chiamata con l'indice 0 (radice) o quando l'elemento è inferiore al padre
- La complessità è $O(\log n)$ perché ogni chiamata risale di un livello

14.1.2 Estrazione

- restituisci il primo elemento dell'array
- metti l'ultimo elemento al posto della radice e decrementa last
- fai scendere l'elemento tramite scambi padre-figlio per mantenere la proprietà dello heap

```
int Heap::extract() {
    int r=h[0];
    h[0]=h[last--];
    down(0);
    return r;
}
```

Funzione down:

```
void Heap::down(int i) { // i e' l'indice dell'elemento da far scendere
    int son=2*i+1; // son = indice del figlio sinistro (se esiste)
    if (son == last) { // se i ha un solo figlio (e' l'ultimo dell'array)
        if (h[son] > h[i]) // se il figlio e' maggiore del padre
            exchange(i,last); // fai lo scambio, altrimenti termina
    }
    else if (son < last) { // se i ha entrambi I figli
        if (h[son] < h[son+1]) son++; // son= indice del maggiore fra i due
        if (h[son] > h[i]) { // se il figlio e' maggiore del padre
            exchange(i,son); // fai lo scambio
            down(son); // e chiama down sulla nuova posizione
        } // altrimenti termina (termina anche se i non ha figli)
    }
}
```

complessità = $O(\log n)$

Lo heap è particolarmente indicato per l'implementazione del tipo di dato astratto coda con priorità.

Si tratta di una coda in cui gli elementi contengono, oltre all'informazione, un intero che ne definisce la priorità.

In caso di estrazione, l'elemento da estrarre deve essere quello con maggiore priorità.

14.2 Algoritmo Heapsort

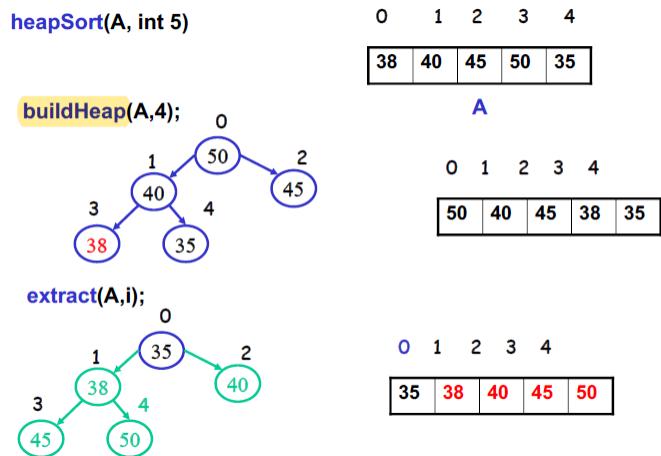


Figura 14.1: Al momento della buildHeap ci si trova in un passo intermedio: l'array non è ancora ordinato ma è "a forma" di heap

Definizione 40 Algoritmo di ordinamento Heapsort: .

- trasforma l'array in uno heap (buildheap)
- esegui n volte l'estrazione scambiando ogni volta il primo elemento dell'array con quello puntato da last

```
void heapSort(int* A, int n) { // n dimensione
    dell array
    buildHeap(A,n-1); // O(n)
    int i=n-1;
    while (i > 0) { // O(nlogn)
        extract(A,i);
    }
} //O(nlogn)
```

14.2.1

Funzione down modificata: (si chiama down perché va dalla radice verso il basso).

I parametri sono: l'array, l'indice dell'elemento da far scendere, l'ultimo elemento dello heap

```
void down(int * h, int i, int last) {
    int son=2*i+1;
    if (son == last) {
```

```

        if (h[son] > h[i]) exchange(h, i, last);
    }
    else if (son < last) {
        if (h[son] < h[son+1]) son++;
        if (h[son] > h[i]) {
            exchange(h, i, son);
            down(h, son, last);
        }
    }
}
//O(logn)

```

14.2.2 Trasforma l'array in uno heap (buildHeap)

- Esegui la funzione down sulla prima metà degli elementi dell'array (gli elementi della seconda metà sono foglie)
- Esegui down partendo dall'elemento centrale e tornando indietro fino al primo

```

void buildHeap(int* A, int n) { //n+1 è la dimensione dell'array
    for (int i=n/2; i>=0; i--) down(A, i, n);
}
//O(n)

```

14.2.3 Extract modificata

- I parametri sono l'array e l'ultimo elemento dello heap
- L'ultimo elemento viene scambiato con il primo
- Non si restituisce nulla
- complessità = O(logn)

```

void extract(int* h, int & last) {
    exchange(h, 0, last--);
    down(h, 0, last);
}

```

14.2.4 risultato finale

```
void down(int * h, int i, int last) {
    int son=2*i+1;
    if (son == last) {
        if (h[son] > h[i]) exchange(h, i, last);
    } else if (son < last) {
        if (h[son] < h[son+1]) son++;
        if (h[son] > h[i]) {
            exchange(h, i, son);
            down(h, son, last);
        }
    }
}

void extract(int* h, int & last) {
    exchange(h, 0, last--);
    down(h, 0, last);
}

void buildHeap(int* A, int n) {
    for (int i=n/2; i>=0; i--) down(A,i,n);
}

void heapSort(int* A, int n) {
    buildHeap(A,n-1);
    int i=n-1;
    while (i > 0) extract(A,i);
}
```

Capitolo 15

Alberi Binari di Ricerca, Gestione Stringhe, Progettazione (Lezione 9 Virdis)

Funzioni min e max

```
Node * min()
{
    Node * temp = root_;
    while( temp->left != NULL )
        temp = temp->left;
    return temp;
}

Node * max()
{
    Node * temp = root_;
    while( temp->right != NULL )
        temp = temp->right;
    return temp;
}
```

Funzione height

```
int height( Node * tree )
{
    int hLeft;
    int hRight;
    if( tree == NULL )
        return 0;
    hLeft = height(tree->left);
    hRight = height(tree->right);
    return 1 + max(hLeft,hRight);
}
```

Funzione cerca ma booleana:

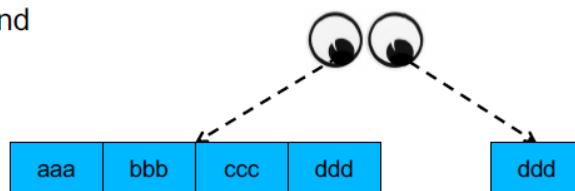
```

bool search( Node * tree , int val )
{
    if( tree == NULL )
        return false;
    bool found;
    if( tree->value == val )
        return true;
    else if( val <= tree->value )
        found = search( tree->left , val );
    else
        found = search( tree->right , val );
    return found;
}

```

15.1 Stringhe

- Creazione
- Concatenazione
- Compare
- Find



```

#include <string>
String parola = "liste";
String frase = "mi piacciono le liste";
String parola2 = "non ";
String frase2 = parola2 + frase;
frase.find(parola);
// se fallisce -> string::npos
parola.compare(parola2);

```

Esercizio Stringhe

- Input
 - Una testo T formato da più parole
 - Un insieme S di N parole
- Output:
 - 1.Le parole di S *contenute* in T, ordinate per posizione in T (insieme R1)
 - 2.Le parole di S *non contenute* in T, in ordine lessicografico (insieme R2)

Implementazione:

```
// cerca stringhe di S dentro T
void trovaInT( ... ){ }
// implementa confronto per posizione
bool compare1(string a, string b){ }
// implementa confronto lessicografico
bool compare2(string a, string b){ }
int main()
{
    string T;
    vector <string> S, R1, R2;
    // lettura T ed S
    trovaInT( ... );
    sort( R1.begin(), R1.end(), compare1 )
    sort( R2.begin(), R2.end(), compare2 )
    print();
}
```

es sulla ricerca (già vista nei capitoli precedenti)

Capitolo 16

Limiti Inferiori, Alberi di Decisione, Algoritmi di ordinamento (Lezione 7 Ducange

16.1 Limiti inferiori

Definizione 41 Limite inferiore: Un problema è di ordine $\Omega(f(n))$ se non è possibile trovare un algoritmo che lo risolva con complessità minore di $f(n)$. Tutti gli algoritmo che risolvono il problema sono $O(f(n))$

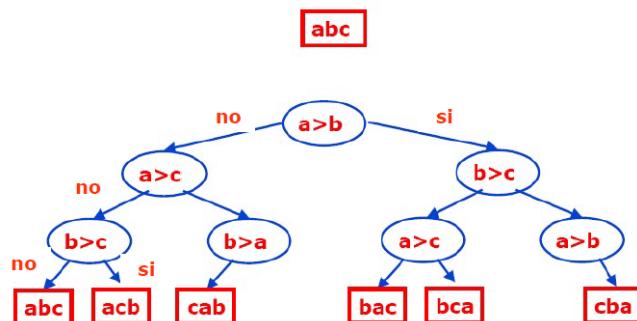
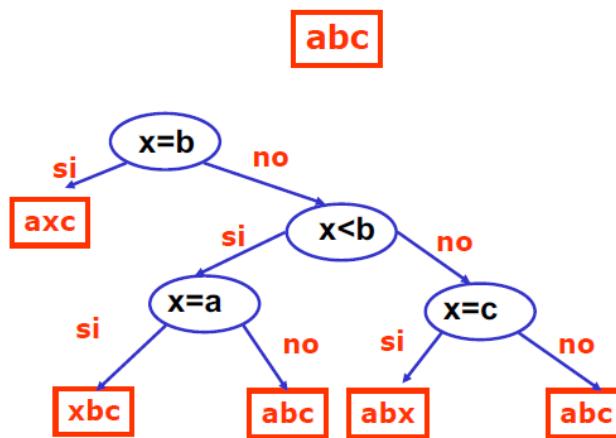
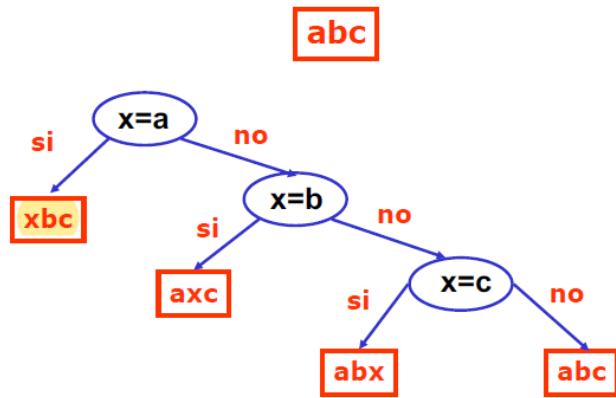
16.2 Alberi di decisione

Si applicano soltanto agli algoritmi:

- basati su confronti (la complessità è proporzionale al numero di confronti che facciamo)
- che hanno complessità proporzionale al numero di confronti che vengono effettuati durante l'esecuzione dell'algoritmo

Definizione 42 Albero di decisione: def appunti: è una albero binario che nella sua esplorazione viene utilizzato per la risoluzione di una problema
def slide: albero binario che corrisponde all'algoritmo:

- ogni foglia rappresenta una soluzione per un particolare assetto dei dati iniziali
- ogni cammino dalla radice ad una foglia rappresenta una esecuzione dell'algoritmo (sequenza di confronti) per giungere alla soluzione relativa alla foglia



Ogni algoritmo che risolve un problema che ha s soluzioni ha un albero di decisione corrispondente con almeno s foglie.

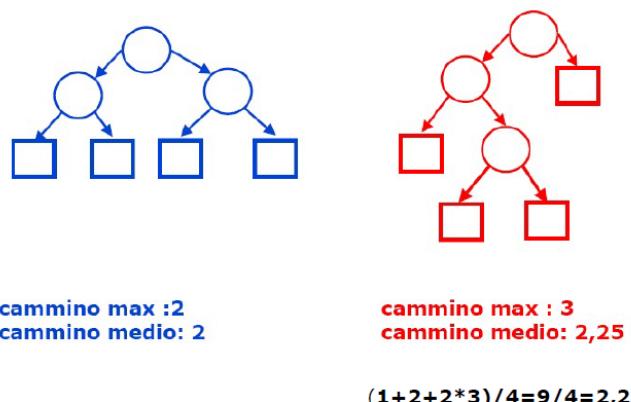
Fra tutti gli alberi di decisione per un particolare problema che ha s soluzioni:

- l'albero di decisione che minimizza la lunghezza massima dei percorsi fornisce un limite inferiore al numero di confronti che un algoritmo che risolve il problema deve fare nel caso peggiore.

- l'albero di decisione che minimizza la lunghezza media dei percorsi fornisce un limite inferiore al numero di confronti che un algoritmo che risolve il problema deve fare nel caso medio

Fatti:

- Un albero binario con k livelli ha al massimo 2^k foglie (ce l'ha quando è bilanciato)
- un albero binario con s foglie ha almeno $\log_2 s$ livelli
- gli alberi binari bilanciati minimizzano sia il caso peggiore che quello medio: hanno $\log s(n)$ livelli



16.3 Algoritmi di ordinamento

Numero di soluzioni: $n!$

Cammino medio e max: $\log(n!) \cong n \log(n)$

- MergeSort è ottimo
- quicksort è ottimo nel caso medio
- non sempre il limite è raggiungibile (la ricerca è $\Omega(\log n)$)

Ordinamenti con complessità minore di $O(n \log n)$:

- counting sort
- radiz sort

16.3.1 Counting sort

- Ordina una sequenza di interi
- Si può usare quando si conoscono i valori minimo e massimo degli elementi da ordinare
- Per ogni valore presente nell'array, si contano gli elementi con quel valore utilizzando un array ausiliario avente come dimensione l'intervallo dei valori
- Successivamente si ordinano i valori tenendo conto dell'array ausiliario

A=	<table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td></tr> <tr><td>7</td><td>7</td><td>4</td><td>4</td><td>7</td><td>5</td><td>4</td><td>7</td><td>4</td><td>5</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> </table>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	7	7	4	4	7	5	4	7	4	5	1	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13																
7	7	4	4	7	5	4	7	4	5	1	1	0	1																
C=	<table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr> <tr><td>1</td><td>3</td><td>0</td><td>0</td><td>4</td><td>2</td><td>0</td><td>4</td></tr> </table>	0	1	2	3	4	5	6	7	1	3	0	0	4	2	0	4												
0	1	2	3	4	5	6	7																						
1	3	0	0	4	2	0	4																						
	n=14 minimo=0 massimo=7 possibili valori:8																												
	<table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>4</td><td>4</td><td>4</td><td>4</td><td>5</td><td>5</td><td>7</td><td>7</td><td>7</td><td>7</td></tr> </table>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	0	1	1	1	4	4	4	4	5	5	7	7	7	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13																
0	1	1	1	4	4	4	4	5	5	7	7	7	7																

```
void counting_sort(int A[], int k, int n){
    // 0 è il minimo, k il massimo, C array ausiliario
    int i, j; int C[k+1];
    for (i=0; i<=k; i++) C[i] = 0; // O(k)
    for (j=0; j<n; j++) C[A[j]]++; // O(n)
    j=0;
    for (i=0; i<=k; i++) // O(?)
        while (C[i]>0){
            A[j]=i;
            C[i]--;
            j++;
        }
}
```

Proprietà:

- non è basato su confronti
- complessità $O(n+k)$ (nel caso sopra citato $O(n+8)$)
- conviene quando k è $O(n)$
- Necessaria memoria ausiliaria

16.3.2 Radix Sort

- Ordina una sequenza di interi
- Si può usare quando si conosce la lunghezza massima (numero di cifre) d dei numeri da ordinare
- Si eseguono d passate ripartendo, in base alla d -esima cifra, i numeri in k contenitori, dove k sono i possibili valori di una cifra, e rileggendo il risultato con un determinato ordine

esempio radix sort con cifre decimali:

Numeri da ordinare ($d=3, k=10$): 190, 051, 054, 207, 088, 010

1° passata ($O(n+k)$)

Si inseriscono i numeri nei contenitori in base al valore dell'ultima** cifra (la meno significativa)**

010									
190	051			054			207	088	
0	1	2	3	4	5	6	7	8	9

Si estraggono i numeri rileggendoli da sinistra a destra e dal basso verso l'alto:

190, 010, 051, 054, 207, 088

190, 010, 051, 054, 207, 088

2° passata ($O(n+k)$)

Si inseriscono i numeri nei contenitori in base al valore della **penultima cifra**

					054				
207	010				051			088	190
0	1	2	3	4	5	6	7	8	9

Si estraggono i numeri rileggendoli da sinistra a destra e dal basso verso l'alto:

207, 010, 051, 054, 088, 190

207, 010, 051, 054, 088, 190

3° e ultima passata ($O(n+k)$)

Si inseriscono i numeri nei contenitori in base al valore della prima cifra

088									
054									
051									
010	190	207							
0	1	2	3	4	5	6	7	8	9

Si estraggono i numeri rileggendoli da sinistra a destra e dal basso verso l'alto:

010, 051, 054, 088, 190, 207

Proprietà:

- Non basato su confronti
- E' fondamentale partire dalla cifra meno significativa
- La complessità è $O(d(n+k))$ dove d è la lunghezza delle sequenze e k è il numero dei possibili valori di ogni cifra (nel caso dell'esempio $O(3(n+10))$)
- Necessaria memoria ausiliaria
- Conveniente quando d è molto minore di n
- Si può usare per ordinare in ordine alfabetico sequenze di caratteri

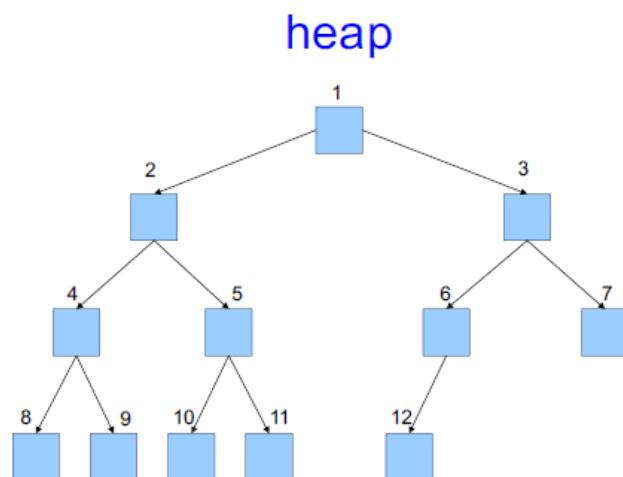
```
procedura bucketSort(array A di n interi, interi b e t)
1.   sia Y un array di dimensione b
2.   for i = 1 to b do Y[i] ← lista vuota
3.   for i = 1 to n do
4.       c ← t-esima cifra di A[i] nella rappresentazione in base b
5.       appendi A[i] alla lista Y[c + 1]
6.   for i = 1 to b do
7.       copia ordinatamente in A gli elementi della lista Y[i]

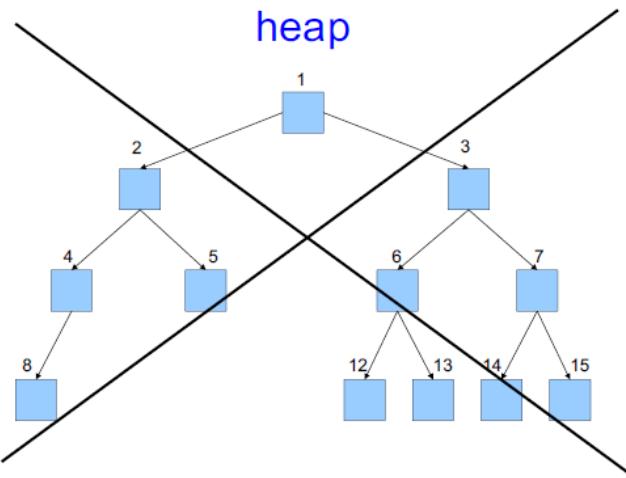
algoritmo radixSort(array A di n interi)
8.   t ← 0
9.   while (esiste un numero la cui t-esima cifra è ≠ 0)
10.      bucketSort(A, 10, t)
11.      t ← t + 1
```

Capitolo 17

Heap (Lezione 10 Virdis)

ALberi binari in cui tutti i livelli hanno il massimo numero di nodi tranne al peggio l'ultimo e in cui le foglie sono raggruppate verso sinistra





Si può trasformare uno heap in un vettore e non tanto un albero binario perché quest'ultimo potrebbe avere dei genitori senza figli a livelli intermedi (che andrebbero rappresentati con un puntatore a null e complicherebbero la questione)

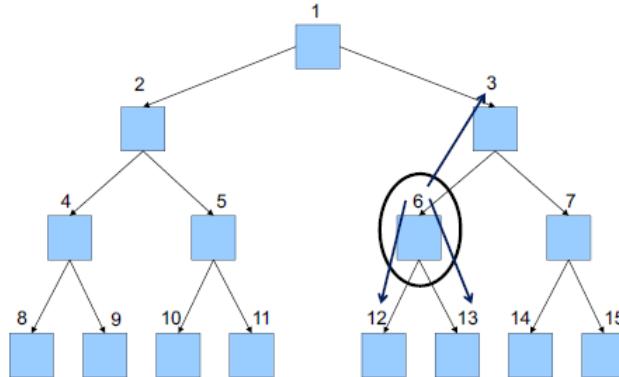
17.1 Classe heap

```

class Heap
{
    std::vector<int> data_;
    int length_; // lunghezza array
    int size_; // dimensione Heap
public:
    Heap() {};
    void fill( int l );
    void printVector();
    ...
}

```

size e length coincidono solo se lo heap è in uno stato corretto



17.1.1 Funzioni relative all'immagine 17.1:

```

int parent(int i)
{
    return floor((i-1)/2); // floor(i/2)
}
int getLeft(int i)
{
    return (i*2) + 1; // i*2
}
int getRight(int i)
{
    return (i*2)+2; // (i*2)+1
}

```

note sul codice precedente:

- la funzione floor esplicita l'arrotondamento all'intero inferiore evitando ambiguità
- il codice commentato è l'implementazione per heap che partono da indice 1

17.1.2 Stampa con la grafica ganza

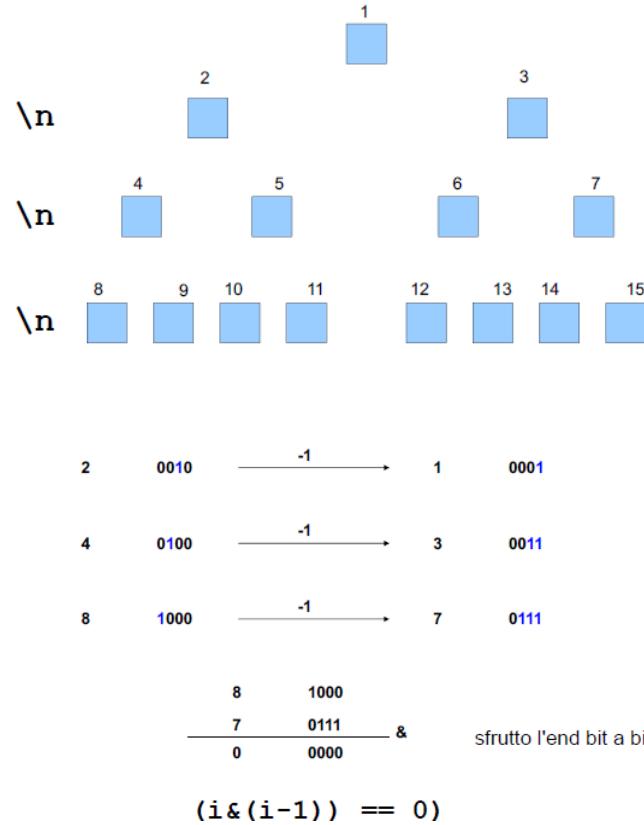


Figura 17.1: ricerca delle potenze di due tramite la rappresentazione binaria delle potenze di 2 (tutti 0 tranne un 1)

```

bool isFirstChild( int i )
{
    if( ( i!=0 ) && ( (i&(i-1)) == 0) )
        return true;
    else
        return false;
}
void print()
{
    for( int i=0 ; i < length_ ; ++i )
    {
        if( isFirstChild(i+1) )
            cout << endl;
        cout << data_[i] << "\t";
    }
    cout << endl;
}

```

}

17.1.3 heap property

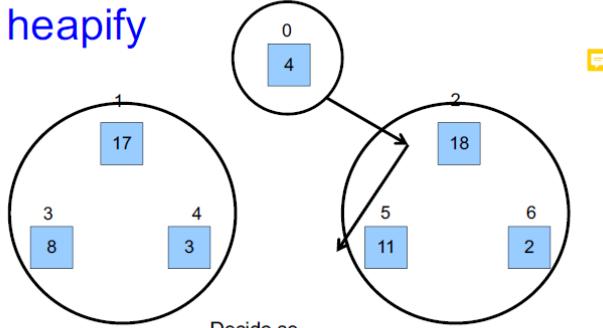
Lo heap ha delle proprietà sui dati che esso contiene, in particolare uno heap deve soddisfare la heap property, in particolare se consideriamo un max heap (come possiamo anche considerare un min heap che però avrà la proprietà duale) (si dice infatti proprietà di maxheap) ogni nodo dell'albero avrà etichetta maggiore delle etichette dei suoi figli. Quindi la heap property è una proprietà che deve essere mantenuta dall'albero binario (oltre a quelle strutturali) per essere definito uno heap.

ATTENZIONE: questa prop non dice nulla sull'ordinamento globale dell'albero, ci da solo una info sull'ordinamento parziale.

17.1.4 Funzione heapify

Dato un nodo e due sottoalberi che soddisfano la proprietà di max heap (o min a seconda delle necessità) sistema il nodo considerato all'inizio nella posizione corretta dello heap.

Esempio heapify



- Decido se
 - già ok?
 - andare a destra
 - andare a sinistra

```

void maxHeapify(int i)
{
    // ottengo left e right

    // (se ho figlio left) AND (left > i)
    // left è più grande
    // altrimenti
    // i è più grande

    // (se ho figlio right) AND (right > largest)
    // right è più grande

    // se i viola la proprietà di max-heap
    {
        // scambio i e il più grande
        // controllo se l'albero che ho cambiato va bene
    }
}

```

Figura 17.2: pseudocode

```

void maxHeapify(int i)
{
    int left = getLeft(i);
    int right = getRight(i);
    int largest;
    if((left < size_)&&(data_[left] > data_[i]))
        largest = left;
    else
        largest = i;
    if((right < size_)&&(data_[right] > data_[largest]))
        largest = right;
    if( largest != i )
    {
        scambia(i,largest);
        maxHeapify(largest);
    }
}

```

17.1.5 Funzione build heap (MaxHeap)

```

void buildMaxHeap()
{
    size_ = length_;
    int i = floor(length_/2)-1;
    for( ; i>=0 ; --i )
    {
        maxHeapify(i);
        print();
    }
}

```

int i = //indice del primo nodo con figli -> primo nodo che potrebbe non soddisfare la proprietà di maxheap in quanto le foglie la soddisfano a prescindere.

Utilizzo: la proprietà di maxheap dà solo informazioni sulla relazione tra padre che è maggiore dei figli

17.1.6 HeapSort

Dall'array in cui è stato inserito lo heap prendo il massimo (la radice) e lo scambio con l'ultimo, ho quindi un maxheap preceduto dal primo elemento (che era l'ultimo) e succeduto dalla ex radice, applico quindi la maxheapify sull'array che ha indice finale last-1 e riparto con lo scambio ecc.

Alla fine l'array è ordinato il modo decrescente.

17.1.7 Heap STL

```
#include <algorithm>
make_heap( inizio , fine )//=build heap
pop_heap( inizio , fine )//prende il primo elemento lo sposta alla fine e ri
#include <queue>
priority_queue<int> prioQ
prioQ.push(val)
prioQ.top()//ritorna il valore più grande
prioQ.pop()//elimina il valore più grande
```

Poi ci sono tanti esempi vari **VEDI PDF**

Capitolo 18

Hash (Lezione 8 Ducange)

Definizione 43 Funzione hash: $h(\text{funzione hash})$: InfoType \rightarrow indici
è una funzione suriettiva

$$x \rightarrow h(x)$$

con $n \leq k$ dati:

- n = numero massimo di elementi (non rappresenta però tutti i possibili elementi di uno specifico dominio (es dominio dei numeri naturali = infinito))
- k = dimensione dell'array

Se si pone h anche iniettiva risolve il problema dell'accesso diretto (e escluderebbe collisioni) ma non risolverebbe il problema della memoria dato che k può essere molto grande.

Mantenere l'iniettività risulta però una follia in caso di domini grandissimi, subentrano quindi i problemi di collisioni

$h(x)$: indirizzo hash dell'elemento che contiene x

```
bool hashSearch (InfoType *A , int n, InfoType x) {  
    int i=h(x); if (A[i]==x) return true;  
    else return false;  
}
```

Problema: memoria (k può essere molto grande)

La complessità è legata al calcolo della hash che in questo caso è unitario quindi la complessità è costante

Insieme= $\{0,1,2,7,9\}$ (non noto apriori)

n= 5
k= 10

Inserimenti: { 0, 2, 7 } **h(0)= 0**
■ **h(2)=2**
h(x)= x **h(7)=7**

NB: non è necessario memorizzare l'elemento

0	1
1	0
2	1
3	0
4	0
5	0
6	0
7	1
8	0
9	0

Quando gli elementi del dominio sono molti si rilascia l'iniettività e si permette che due elementi diversi abbiano lo stesso indirizzo hash:

$$h(x_1) = h(x_2) \text{ collisione}$$

Bisogna quindi gestire le seguenti situazioni:

- come si cerca un elemento se si trova il suo posto occupato da un altro
- come si inseriscono gli elementi

18.1 Metodo hash ad indirizzamento aperto (non si fa uso di puntatori(?))

18.1.1 Funzione hash modulare

Definizione 44 Funzione hash modulare: $h(x)=(x \% k)$ dove k è la dimensione dell'array

(siamo sicuri che vengono generati tutti e soli gli indici dell'array)

Definizione 45 Legge di scansione lineare: se non si trova l'elemento al suo posto, lo si cerca nelle posizioni successive fino a trovarlo o ad incontrare una posizione vuota (def appunti non molto chiara direi: legge di scansione: ne esistono di vari tipi ma sostanzialmente in generale prende il valore che voglio inserire e un indice che indica il numero di volte che ho provato a inserirlo)

L'inserimento è fatto con lo stesso criterio.

Una volta inseriti nella struttura dati, non possano più essere cancellati.

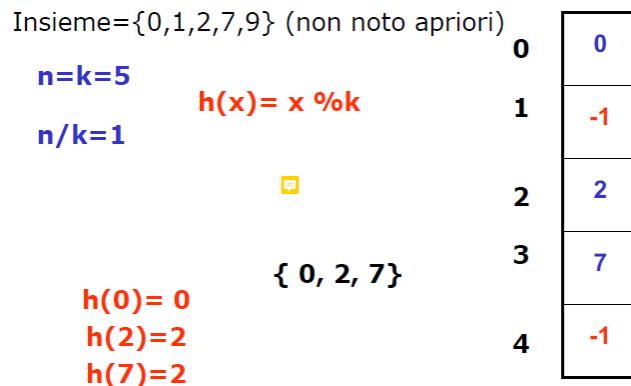
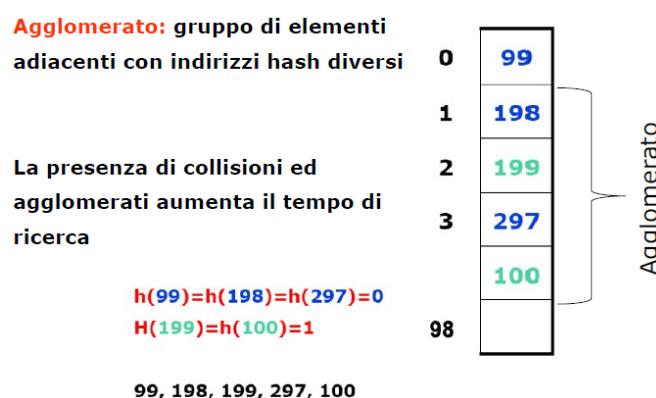
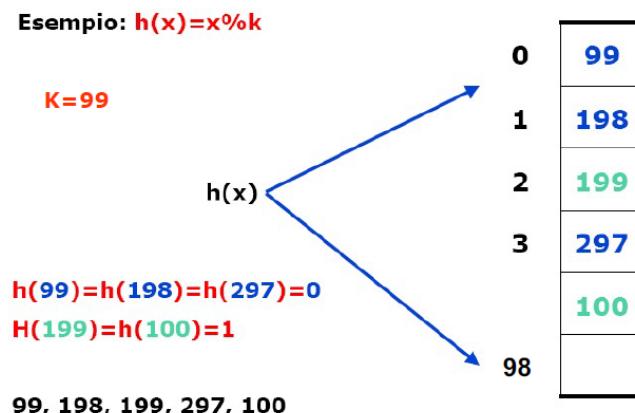


Figura 18.1: quando il vettore è vuoto metto tutti i valori a -1



```
bool hashSearch (int *A, int k, int x) {
    int i=h(x);
    for (int j=0; j<k; j++) {
```

```

        int pos = (i+j)%k; // somma in modulo
        if (A[pos] == -1) return false ;
        if (A[pos] == x) return true ;
    }
    return false ;
}

```

```

int hashInsert (int *A , int k, int x) {
    int i=h(x); int b=0;
    for (int j=0; !b & j<k; j++) {
        int pos = (i+j)%k;
        if (A[pos] == -1) {
            A[pos] = x; b=1;
        }
    }
    return b;
}

```

Consideriamo un array di dimensione 10 che dovrà contenere numeri naturali, con funzione hash $h(x) = x \% 10$.

Supponiamo che nell'array siano stati inseriti, nell'ordine, i numeri 94, 52, e 174. La situazione sarà quella di Figura (a):

	0	1	2	3	4	5	6	7	8	9
(a)	-1	-1	52	-1	94	174	-1	-1	-1	-1
	0	1	2	3	4	5	6	7	8	9
(b)	-1	-1	52	-1	-2	174	-1	-1	-1	-1

Se effettuassimo la cancellazione dell'elemento 94, allora dovremmo passare alla situazione della Figura (b). Perché?

```

// inserimento in presenza di cancellazioni
int hashInsert (int *A , int k, int x) {
    int i=h(x); int b=0;
    for (int j=0; !b && j<k; j++) {
        int pos = (i+j)%k;
        if ((A[pos] == -1) || (A[pos] == -2)) { // -2=posizione disponibile
            A[pos] = x;
            b=1;
        }
    }
    return b;
}

```

scansione_lineare(x;j) = (h(x) + cost*j) mod k

Es: (h(x) + j) mod k, j=1, 2, ...

scansione_quadratica(x; j) = (h(x) + cost*j^2) mod k

Es: (h(x) + j^2) mod k j=1, 2, ...

Figura 18.2: La diversa lunghezza del passo di scansione riduce gli agglomerati, ma è necessario controllare che la scansione visiti tutte le possibili celle vuote dell'array, per evitare che l'inserimento fallisca anche in presenza di array non pieno.

18.1.2 Tempo medio di ricerca per l'indirizzamento aperto

Il tempo medio di ricerca (numero meido di confronti) dipende da:

- **fattore di carico:** rapporto $\alpha = n/k$ (sempre ≤ 1): numero medio di elementi per ogni posizione
- **Legge di scansione:** (migliore con la scansione quadratica e altre più sofisticate)
- Uniformità della funzione hash (genera gli indici con uguale probabilità)

$\frac{n}{M}$	scansione lineare	scansione quadratica
10%	1.06	1.06
50%	1.50	1.44
70%	2.16	1.84
80%	3.02	2.20
90%	5.64	2.87

$$M=k$$

Figura 18.3: man mano che il fattore di carico aumenta l'efficienza della scansione lineare diminuisce rispetto a quella quadratica

18.1.3 Stima del numero medio di accessi

Numero medio di accessi con la scansione lineare:

$$\leq 1/(1 - \alpha)$$

per esempio per $\alpha = 0,5$ abbiamo 2 accessi, per $\alpha = 0,9$ ne abbiamo 10

18.1.4 Problemi con l'indirizzamento aperto

Molti inserimenti e cancellazioni defradano il tempo di ricerca a causa degli agglomerati. È necessario periodicamente "risistemare" l'array.

18.2 Metodo di concatenazione

- array A di $k \leq n$ puntatori ($n/k \geq 1$) con k che non è più lo spazio che ho a disposizione ma è il primo livello di riferimento dove voglio inserire un'informazione
- elementi che collidono su i nella lista di puntatore $A[i]$
- evita del tutto gli agglomerati

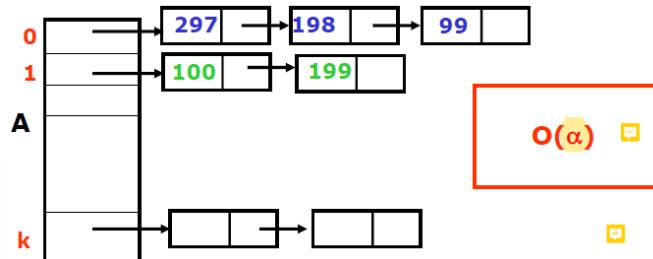


Figura 18.4: complessità per la ricerca $O(\alpha)$ con $\alpha = n/k$ ovvero complessità (nel caso medio) lineare perché l'accesso avviene con complessità costante mentre la ricerca che scorre la lista è lineare

con questo metodo si rischia di avere poche liste lunghe e tanti buchi \rightarrow a questo punto converrebbe un vettore

18.3 Dizionari (tabelle)

chiave	informazione	Ricerca Inserimento Cancellazione

Figura 18.5: con $h(\text{chiave})$ si raggiunge l'informazione

Da Data Base Relazionali a Dizionari K-V

employee_id	first_name	last_name	address
1	John	Doe	New York
2	Benjamin	Button	Chicago
3	Mycroft	Holmes	London

FOREIGN KEY

payment_id	employee_id	amount	date
1	1	50,000	01/12/2017
2	1	20,000	01/13/2017
3	2	75,000	01/14/2017
4	3	40,000	01/15/2017
5	3	20,000	01/17/2017
6	3	25,000	01/18/2017

Capitolo 19

Programmazione dinamica e Algoritmi Greedy (Lezione 9 Ducange)

Si può usare quando non è possibile applicare il metodo del divide et impera (non si sa con esattezza quali sottoproblemi risolvere e non è possibile partizionare l'insieme in sottoinsiemi disgiunti)

Definizione 46 Programmazione dinamica: metodo: si risolvono tutti i sottoproblemi a partire dal basso e si conservano i risultati ottenuti per poterli usare successivamente. (strategia bottom-up)

La complessità del metodo dipende dal numero dei sottoproblemi

Quando si può applicare:

- **Sottostruttura ottima:** una soluzione ottima del problema contiene la soluzione ottima dei sottoproblemi (in altre parole la soluzione ottima del problema è ottenuta a partire dalla soluzione ottima del sottoproblema)
- **sottoproblemi comuni:** un algoritmo ricorsivo richiederebbe di risolvere lo stesso sottoproblema più volte

19.1 Più lunga sottosequenza comune

$$\alpha = \alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5 \alpha_6 \alpha_7$$

$$\beta = \beta_1 \beta_2 \beta_3 \beta_4 \beta_5 \beta_6$$

$$\alpha = a \ b \ c \ a \ b \ b \ a \quad \beta = c \ b \ a \ b \ a \ c$$

3 PLSC: **baba, cbba, caba**

Lunghezza delle PLSC = 4

$$\alpha = \alpha_1 \dots \alpha_i \dots \alpha_m \quad \beta = \beta_1 \dots \beta_j \dots \beta_n$$

L(i,j) = lunghezza delle PLSC di $\alpha_1 \dots \alpha_i$ e $\beta_1 \dots \beta_j$

$$L(0,0)=L(i,0)=L(0,j)=0$$

$$L(i,j)=L(i-1,j-1)+1 \quad \text{se } \alpha_i = \beta_j$$

$$L(i,j)=\max(L(i,j-1), L(i-1,j)) \quad \text{se } \alpha_i \neq \beta_j$$

Sequenze originali:

$$\alpha = \alpha_1 \dots \alpha_i \dots \alpha_m \quad \beta = \beta_1 \dots \beta_j \dots \beta_n$$

$$\alpha_1 \dots \alpha_i \text{ e } \beta_1 \dots \beta_j$$

con $i \leq n$ e $j \leq m$, sono due **prefissi** di α e β rispettivamente

Se la sequenza γ è una PLSC fra le due sotto-sequenze,
 γ sarà sicuramente un prefisso (primo pezzo) di una PLSC fra α e β completi.

$\alpha = \mathbf{a b c a b b a} \quad \beta = \mathbf{c b a b a c}$

$\alpha_1 = \mathbf{a b c a} \quad \beta_1 = \mathbf{c b a b}$

$\chi = \mathbf{ba, ca}$

3 PLSC: **baba, cbba, caba**

```
int length(char* a, char* b, int i, int j) {
    if (i==0 || j==0) return 0;
    if (a[i]==b[j]) return length(a, b, i-1, j-1)+1;
    else
        return max(length(a,b,i,j-1),length(a,b,i-1,j));
};
```

Relazione di ricorrenza:

$$T(k) = b + 2T(k - 1)$$

La funzione ha un tempo esponenziale in k (minimo fra n e m) -> questo non ci piace -> proviamo a sfruttare la programmazione dinamica e a evitare quella ricorsiva.

Costruire in maniera iterativa tutti gli $L(i,j)$ a partire dagli indici più piccoli (bottom-up) tramite la matrice:

$L(0,0), L(0,1) \dots L(0,n),$

$L(1,0), L(1,1) \dots L(1,n),$

...

$L(m,0), L(m,1) \dots L(m,n)$

```
const int m=7; const int n=6;
int L [m+1] [n+1]; // +1 perche' la prima riga/colonna e' di zeri
int quickLength(char *a, char *b) {
    for (int j=0; j<=n; j++) L[ 0 ] [ j ]=0; // prima riga
    for (int i=1; i<=m; i++) { // i-esima riga
        L[ i ] [ 0 ]=0;
        for (j=1; j<=n; j++)
            if (a[ i ] != b[ j ])
                L[ i ] [ j ] = max(L[ i ] [ j-1 ],L[ i-1 ] [ j ]);
            else L[ i ] [ j ]=L[ i-1 ] [ j-1 ]+1;
```

```

    }
    return L[ m ] [n ];
}

```

Complessità: $O(n*m) =$ complessità polinomiale

	c	b	a	b	a	c
0	0	0	0	0	0	0
a	0	0	0	1	1	1
b	0	0	1	1	2	2
c	0	1	1	1	2	3
a	0	1	1	2	2	3
b	0	1	2	2	3	3
b	0	1	2	2	3	3
a	0	1	2	3	3	4

19.1.1 Estrazione di una PLSC

	c	b	a	b	a	c
0	0	0	0	0	0	0
a	0	0	0	1	1	1
b	0	0	1	1	2	2
c	0	1	1	1	2	3
a	0	1	1	2	2	3
b	0	1	2	2	3	3
b	0	1	2	2	3	3
a	0	1	2	3	3	4

Figura 19.1: cbba

riguardo a 19.1 a e c sono diversi quindi guardo il massimo sulla diagonale che lo precede (4 e 3) guardando prima il valore a sinistra, guardando prima il valore in alto si potrebbero trovare altre sequenze -> se si vogliono trovare tutte le sequenze con questo metodo si va verso la complessità esponenziale

```
void print(int ** L, char *a, char *b, int i=m, int j=n){
    if ((i==0) || (j==0) ) return;
    if (a[i]==b[j]) {
        print(a,b, i-1, j-1);
        cout << a[i];
    }
    else if (L[i][j] == L[i-1][j])
        print(a,b, i-1, j);
    else print(a,b, i, j-1);
}
```

Complessità $n+m$ perché al massimo fa $n + m$ passi dato che una volta decrementa uno e una volta l'altro (potenzialmente) nel caso peggiore

19.2 Algoritmi greedy (Avido/goloso)

La soluzione ottima si ottiene mediante una sequenza di scelte.

In ogni punto dell'algoritmo, viene scelta la strada che in quel momento sembra la migliore.

La scelta locale deve essere in accordo con la scelta globale: scegliendo ad ogni passo l'alternativa che sembra la migliore non si dovrebbero perdere alternative che potrebbero rivelarsi migliori nel seguito. Tuttavia, anche se localmente sembra di aver fatto una scelta migliore, il risultato non è magari ottimale ma potrebbe essere sub-ottimo.

Metodotop-down

19.2.1 codici di compressione

Alfabetto : insieme di caratteri (es: a, b, c, d, e, f) **Codice binario**: assegna ad ogni carattere una stringa binaria **Codifica del testo**: sostituisce ad ogni carattere del testo il corrispondente codice binario. **Decodifica**: ricostruire il testo originario.

Il codice può essere a lunghezza fissa o a lunghezza variabile

	a	b	c	d	e	f
frequenza	45	13	12	16	9	5
Codice a lunghezza fissa	000	001	010	011	100	101
Codice a lunghezza variabile	0	101	100	111	1101	1100

Codifica di abc con codice a lunghezza fissa:

000 001 010(9 bit)

Codifica di abc con codice a lunghezza variabile:

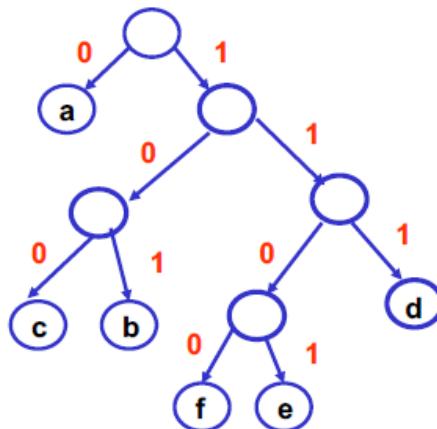
0 101 100(7 bit)

Problema della decodifica

Codice prefisso: nessun codice può essere il prefisso di un altro codice.

I codici prefissi possono essere rappresentati con alberi binari

Rappresentazione ottima: albero pienamente binario



a	b	c	d	e	f	
	0	101	100	111	1101	1100

L'albero ha tante foglie quanti sono i caratteri dell'alfabeto.

L'algoritmo di decodifica trova un cammino dalla radice ad una foglia per ogni carattere riconosciuto

19.2.2 Algoritmo di Huffman

Problema: dato un alfabeto e la frequenza dei suoi caratteri, costruire un codice ottimo (che minimizza la lunghezza in bit delle codifiche)

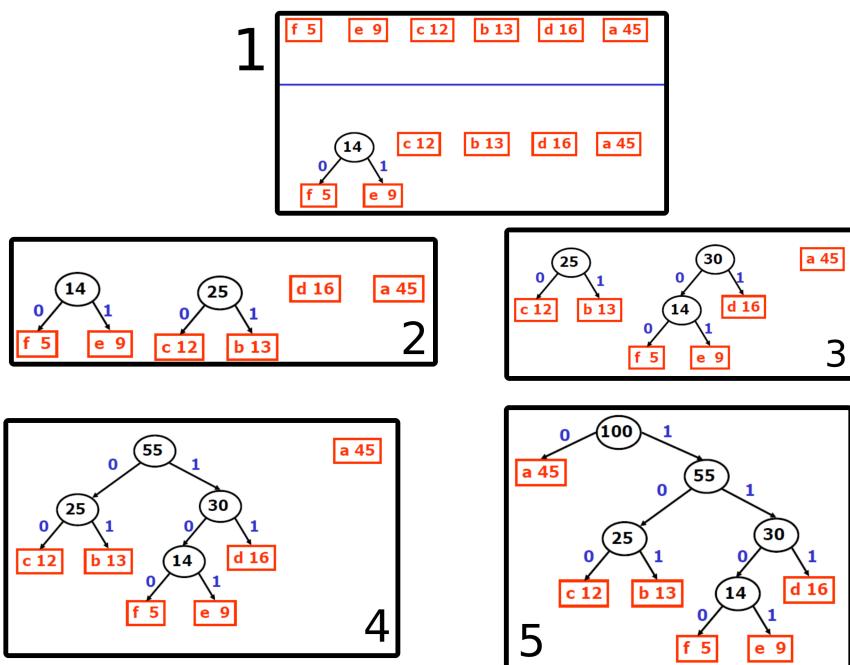
Definizione 47 Algoritmo di Huffman: Costruisce l'albero binario in modo bottom-up

È un algoritmo greedy

Gestisce un foresta di alberi

All'inizio ci sono n alberi di un solo nodo con le frequenze dei caratteri.

Ad ogni passo vengono fusi i due alberi con radice minore introducendo una nuova radice avente come etichetta la somma delle due radici



Complessità

Gli alberi sono memorizzati in un minheap (heap con ordinamento inverso : la radice è il più piccolo)

Si fa un ciclo dove in ogni iterazione:

- vengono estratti i due alberi con radice minore
- vengono fusi in un nuovo albero avente come etichetta della radice la somma delle due radici
- l'albero risultante è inserito nello heap

il ciclo ha n iterazioni ed ogni iterazione ha complessità $O(\log n)$ (si eseguono 2 estrazioni e un inserimento)

$$O(n \log n)$$

perché funziona

La scelta locale è consistente con la situazione globale: sistemando prima i nodi con minore frequenza, questi apparterranno ai livelli più alti dell'albero

```
struct NodeH{
    char symbol; // carattere alfabeto
    int freq; // frequenza carattere
    NodeH* left; NodeH* right;
};
```

```
Node* huffman(Heap H, int n){
    for(int i=0; i< n-1; i++) {
        NodeH *t = new NodeH();
        t->left = H.extract();
        t->right = H.extract();
        t->freq = t->left->freq + t->right->freq; // somma le radici
        H.insert(t); // inserimento nello heap
    }
    return H.extract(); // ritorna la radice dell'albero
}
```

Capitolo 20

Grafi (lezione 10 Ducange)

(grafo = struttura costituita da 2 tipi di elementi)

20.1 Grafi orientati

(anche gli alberi sono particolari tipi di grafi)

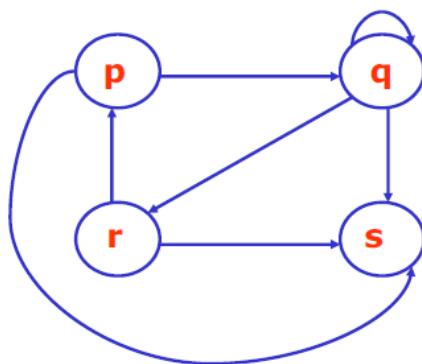
Definizione 48 Grafi orientati: GRAFO ORIENTATO = (N, A)

N = insieme di nodi

$A \subseteq N \times N$ = insieme di archi (coppie ordinate di nodi)

Definizione 49 arco: relazione che si stabilisce tra 2 nodi

Si parla di grafi orientati dal momento in cui gli archi hanno una direzione.



Se $(p,q) \in A$ diciamo che p è predecessore di q e q è successore di p .

$n = |N|$ numero dei nodi

$m = |A|$ numero degli archi

Un grafo orientato con n nodi ha al massimo n^2 archi

Definizione 50 cammino: è una sequenza di nodi (n_1, \dots, n_k) , $k \geq 1$ tale che esiste un arco da n_i a n_{i+1} per ogni $1 \leq i < k$.

La lunghezza del cammino è data dal numero dei archi.

Definizione 51 ciclo: è un cammino che comincia e finisce con lo stesso nodo. Un grafo è aciclico se non contiene cicli.

20.1.1 rappresentazione in memoria dei grafi

Definizione 52 liste di adiacenza: Viene definito un array con dimensione uguale al numero dei nodi. Ogni elemento dell'array rappresenta un nodo con i suoi successori.

```
struct Node{  
    int NodeNumber;  
    Node * next;  
};  
  
Node * graph[H];
```

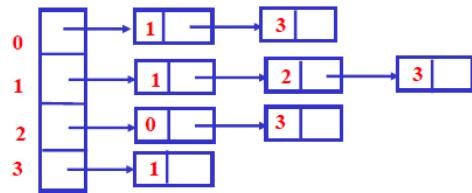


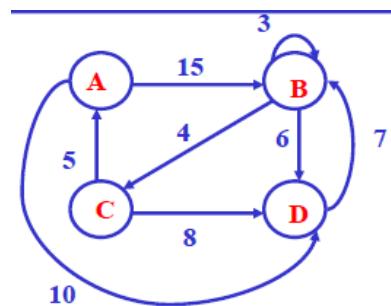
Figura 20.1: ATTENZIONE: considerando la prima lista (quella con testa a indice 0) 3 non è successore di 1; la lista è dei successori di 0

Definizione 53 matrici di adiacenza: nella realizzazione con matrici di adiacenza il grafo viene rappresentato con una matrice quadrata $n \times n$. L'elemento della matrice di indici i, j è 1 se c'è un arco dal nodo i al nodo j e 0 altrimenti.

```
int graph [N] [N];
```

	0	1	2	3
0	0	1	0	1
1	0	1	1	1
2	1	0	0	1
3	0	1	0	0

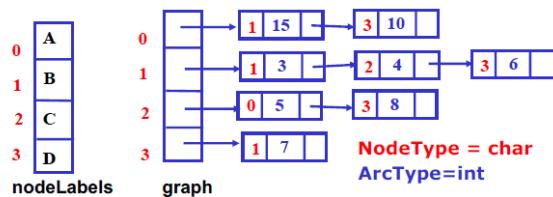
Grafi con nodi e archi etichettati



Liste di adiacenza

```
struct Node{
    int NodeNumber;
    ArcType arcLabel;
    Node * next;
};

Node * graph[N];
NodeType nodeLabels [N];
```



matrici di adiacenza

```
ArcType graph[N][N];
```

```
NodeType nodeLabels[N];
```

	0	1	2	3
0	0	15	0	10
1	0	3	4	6
2	5	0	0	8
3	0	7	0	0

20.1.2 visita in profondità

```
void NodeVisit (nodo) {
    esamina il nodo;
    marca il nodo;
    applica NodeVisit ai successori non marcati del nodo;
}
Void DepthVisit Graph(h) {
    per tutti i nodi:
        se il nodo non e' marcato applica NodeVisit;
}
```

complessità lineare ($n+m$); se non ci fosse il controllo della marcatura dei nodi si andrebbe sul quadratico (n^*m) con $n = \text{numero dei nodi}$ e $m = \text{numero degli archi}$

20.1.3 classe per i grafi

```
class Graph{
    struct Node {
        int nodeNumber;
        Node* next;
    };
    Node* graph [N];
    NodeType nodeLabels [N];
    int mark[N];

    void nodeVisit( int i) {
        mark[i]=1;
        <esamina nodeLabels[i]>;
        Node* g; int j;
        for (g=graph[i]; g; g=g->next){
            j=g->nodeNumber;
            if (!mark[j])    nodeVisit(j);
        }
    }
}
```

```

    }
public:
    void depthVisit() {
        for (int i=0; i<N; i++)
            mark[i]=0;
        for (i=0; i<N; i++)
            if (! mark[i])
                nodeVisit (i);
    }
    ..
};
```

20.2 Grafi non orientati

Definizione 54 grafo non orientato: = (N,A)

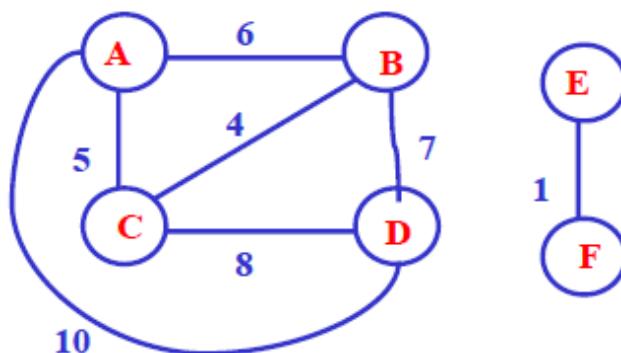
N = insieme di nodi

A = insieme di coppie non ordinate di nodi

Se $(p,q) \in A$, $p \neq q$, diciamo che è adiacente a q e viceversa.

Un grafo non orientato con n nodi ha al massimo $n(n-1)/2$ archi

Esempio:



Un cammino in un grafo non orientato è una sequenza di nodi (n_1, \dots, n_k) , $k \geq 1$ tale che n_i è adiacente a n_{i+1} per ogni i .

Un ciclo è un cammino che inizia e termina con lo stesso nodo e non ha ripetizioni, eccetto l'ultimo nodo.

Un grafo non orientato è connesso se esiste un cammino fra due nodi qualsiasi del grafo.

20.2.1 Rappresentazione in memoria

Un grafo non orientato può essere visto come un grafo orientato tale che, per ogni arco da un nodo p a un nodo q, ne esiste uno da q a p.

La rappresentazione in memoria dei grafi non orientati può essere fatta con le matrici o con le liste di adiacenza tenendo conto di questa equivalenza. Naturalmente ogni arco del grafo non orientato sarà rappresentato due volte (la matrice di adiacenza è sempre simmetrica).

20.3 Multi grafi orientati

Definizione 55 Multi-Grafo: = (N, A)

N =insieme di nodi

A =multi insieme di coppie di nodi

Non c'è relazione fra il numero di nodi e il numero di archi

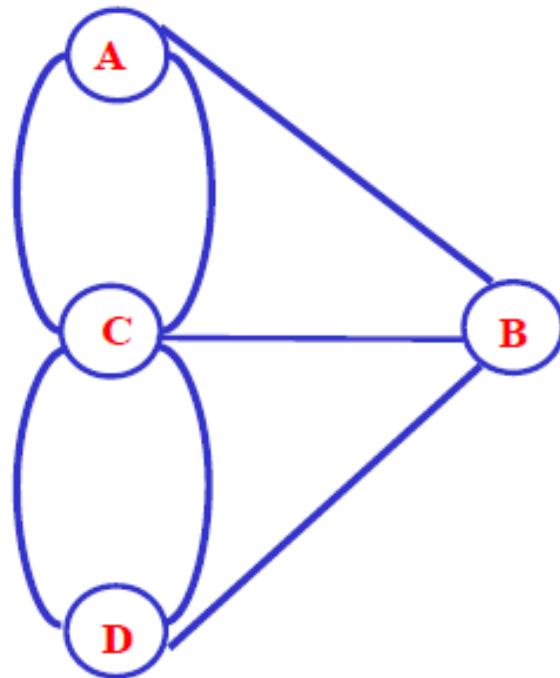
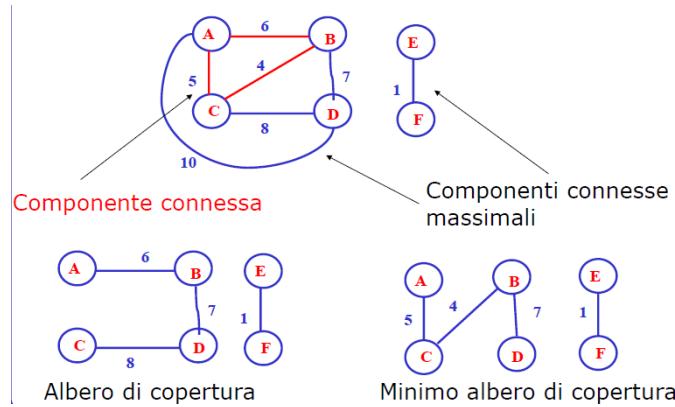


Figura 20.2: esempio di multi grafo NON orientato

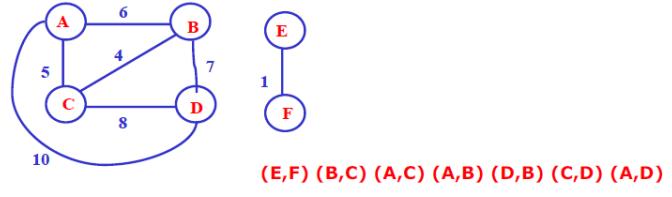
20.4 Minimo albero di copertura



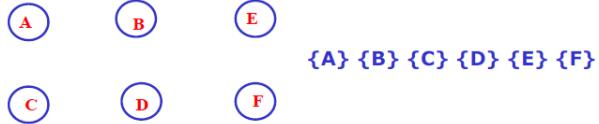
- Un grafo non orientato è connesso se esiste un cammino fra due nodi qualsiasi del grafo
- Componente connessa: sottografo connesso
- Componente connessa massimale: nessun nodo è connesso ad un'altra componente connessa
- Albero di copertura: insieme di componenti connesse massimali acicliche
- Minimo albero di copertura: la somma dei pesi degli archi è minima

20.4.1 algoritmo di Kruskal per trovare il minimo albero di copertura

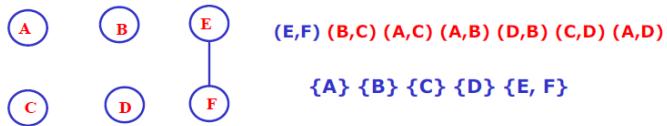
1. Ordina gli archi del grafo in ordine crescente
2. Scorri l'elenco ordinato degli archi:
per ogni arco a
if (a connette due componenti non connesse) {
scegli a;
unifica le componenti;
}



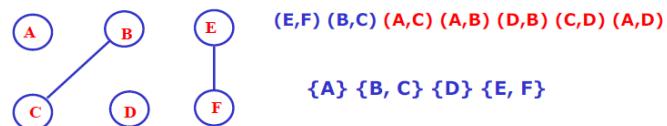
(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)



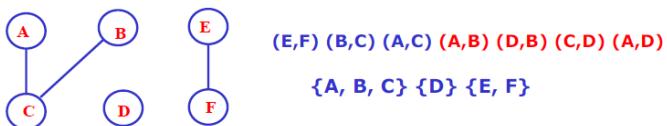
{A} {B} {C} {D} {E} {F}



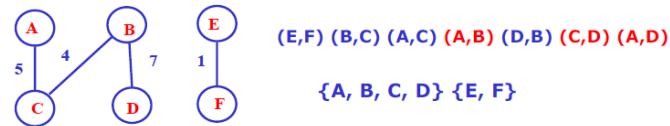
{A} {B} {C} {D} {E, F}



{A} {B, C} {D} {E, F}



{A, B, C} {D} {E, F}



{A, B, C, D} {E, F}

Lunghezza: 17

Capitolo 21

Grafi 2 (Lezione 11 Ducange)

21.1 Algoritmo di Dijkstra

serve per trovare i cammini minimi (esempio navigatori)

Definizione 56 algoritmo di Dijkstra: Si applica ai grafi orientati che hanno peso positivo sugli archi

Trova i cammini minimi da un nodo di partenza a tutti gli altri nodi

Basato sulla metodologia greedy

I cammini minimi, inizialmente posti a “infinito”, vengono aggiornati con stime via via più precise tramite un ciclo nel quale, ad ogni iterazione, un nuovo nodo viene “sistemato”, nel senso che il cammino minimo per lui viene stabilizzato. Alla fine tutti I nodi sono sistematati.

Utilizza due tabelle dist (distanza) e pred (predecessore) con n elementi (n=numero dei nodi)

Per ogni nodo A, dist (A) contiene in ogni momento la lunghezza di un cammino dal nodo iniziale ad A e pred(A) il predecessore di A in questo cammino.

I nodi sono divisi in due gruppi: quelli già sistematati, per i quali i valori delle tabelle dist e pred sono definitivi, e quelli da sistemare (insieme Q). Per i nodi sistematati, dist contiene la lunghezza del minimo cammino e pred permette di ricostruirlo. Per gli altri, dist e pred contengono dati relativi al cammino trovato fino a quel momento, che eventualmente possono essere cambiati se si trova un cammino più corto.

Inizialmente nessun nodo è sistematato: Q contiene tutti i nodi.

Si esegue poi un ciclo. Ad ogni passo:

1. si considera "sistematato" il nodo, fra quelli di Q, con dist minore e lo si toglie da Q

2. si aggiornano pred e dist per gli immediati successori di questo nodo;

Il ciclo termina quando Q contiene un solo nodo.

Pseudo codice: (non vedremo l'implementazione in C++)

```

1 Q = N;
2 per ogni nodo p diverso da p0 { // O(n)
    dist(p)=infinito, pred(p)=vuoto;
}
    dist(p0)=0;
4 while (Q contiene piu di un nodo) {
5 estrai da Q il nodo p con minima dist(p); // O(logn)
6 per ogni nodo q successore di p {
    lpq=lunghezza dell arco (p,q);
    if (dist(p)+lpq < dist(q)) { //O(1)
        dist(q)=dist(p)+lpq; //O(1)
        pred(q)=p; //O(1)
7         re-inserisci in Q il nodo q modificato; //O(logn)
    }
}
}

```

L'insieme Q è memorizzato in un min-heap. Di conseguenza i comandi

```

5 estrai da Q il nodo p con minima dist(p);
7 re-inserisci in Q il nodo q modificato;

```

hanno complessità $O(\log n)$

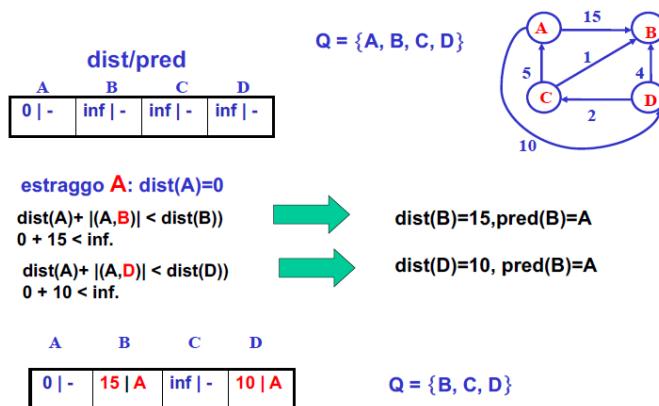
Numero di iterazioni del ciclo while: n

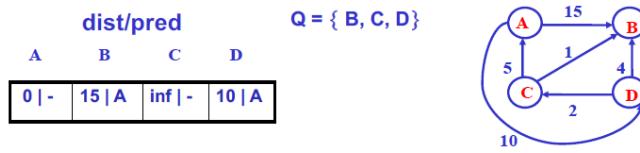
Complessità iterazione: $C[5]+m/n C[7] = O(\log n + (m/n)\log n)$

Complessità dei cicli: $O(n(\log n + (m/n)\log n)) = O(n\log n + m\log n)$ (ATTENZIONE: complessità calcolata con l'utilizzo di un min-heap e in funzione di m e n)

Considerazione: ad ogni iterazione del ciclo i nodi già scelti (eliminati da Q) sono "sistematici": per i nodi già scelti dist contiene la lunghezza del cammino minimo e pred permette di ricostruirlo.

Il cammino minimo per i nodi già scelti passa soltanto da nodi già scelti.

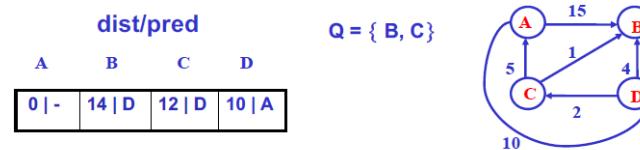




estraggo D: dist(D)=10

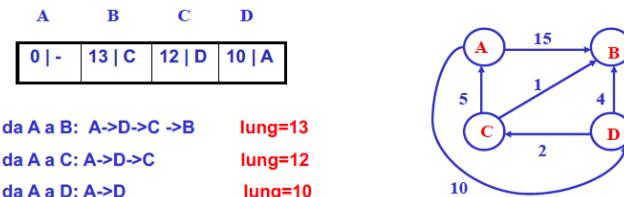
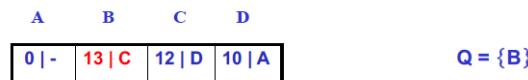
$\text{dist}(D) + |(D, B)| < \text{dist}(B)$ $10 + 4 < 15$ \rightarrow $\text{dist}(B) = 14, \text{pred}(B) = D$

$\text{dist}(D) + |(D, C)| < \text{dist}(C)$ $10 + 2 < \text{inf.}$ \rightarrow $\text{dist}(C) = 12, \text{pred}(C) = D$



estraggo C: dist(C)=12

$\text{dist}(C) + |(C, B)| < \text{dist}(B)$ $12 + 1 < 14$ \rightarrow $\text{dist}(B) = 13, \text{pred}(B) = C$



Nodo scelto	Q	A	B	C	D
	A, B, C, D	0 / -	i / -	i / -	i / -
A	B, C, D	0 / -	15/A	i / -	10/A
D	B, C	0 / -	14/D	12/D	10/A
C	B	0 / -	13/C	12/D	10/A

dist/pred

21.2 Algoritmo PageRank

Cenni su quello di google:

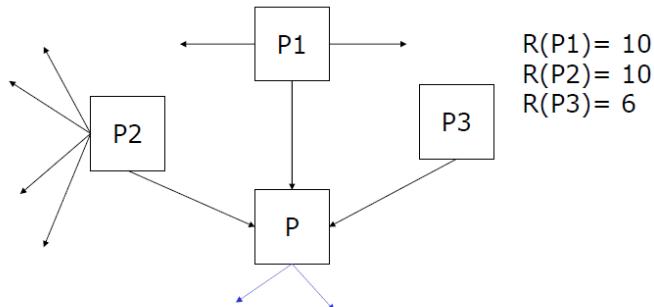
- Serve al motore di ricerca per trovare le pagine web di interesse per una interrogazione.

- Si basa sulle connessioni (link) fra le pagine.
- Considera la rete come un grafo in cui le pagine sono i nodi e i link sono gli archi.
- Calcola il "rango" (rilevanza) di una pagina web P: $R(P)$.
- La rilevanza di P dipende da quanto sono rilevanti le pagine che puntano a P (hanno un link verso P) ma anche da quanti link escono da queste pagine.

Formula base algoritmo PageRank:

$$R(P) = \sum_{Q \rightarrow P} \frac{R(Q)}{|Q|}$$

$|Q|$ = numero di link uscenti da Q
 $Q \rightarrow P$: link da Q a P



$$R(P) = 10/3 + 10/5 + 6/1 = 11,3$$

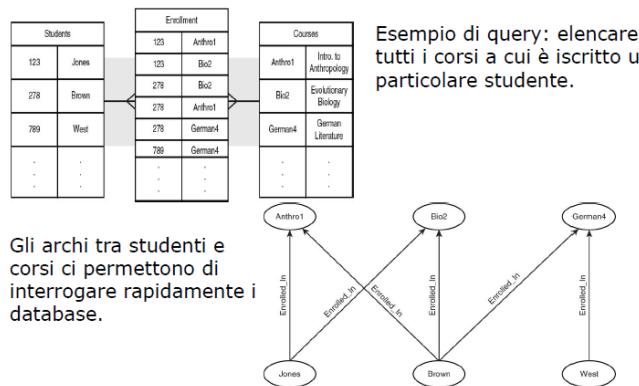
Il calcolo viene fatto calcolando il rango dei nodi in modo iterativo utilizzando la matrice di adiacenza e partendo da un valore del rango uguale per tutti i nodi.

Sono necessari aggiustamenti della formula per assicurarne la convergenza (cicli, nodi pozzo)

21.3 Graph Database

- Adottano vertici e archi per memorizzare relazioni esplicite tra entità.
- Nei database relazionali, le connessioni non sono rappresentate come collegamenti. Invece, due entità condividono un valore di attributo comune, che è noto come una chiave.

- Le operazioni di join sono usate nei database relazionali per trovare connessioni o collegamenti.
- Quando si ha a che fare con un'enorme quantità di dati (diverse tabelle con troppe righe) le operazioni di join diventano computazionalmente costose.



altri 2 esempietti in foto alla fine del pdf

Capitolo 22

NP-Completezza (Lezione 12 Ducange)

spesso ci sono problemi non risolvibili con complessità polinomiale, quindi gli alg spesso sono basati sulla forza bruta ovvero sul controllo di tutte le possibili permutazioni o su un approccio greedy che però da soluzioni sub ottime ma che potrebbero rimanere lontano dall'essere polinomiale.

22.1 Problemi difficili

22.1.1 Zaino

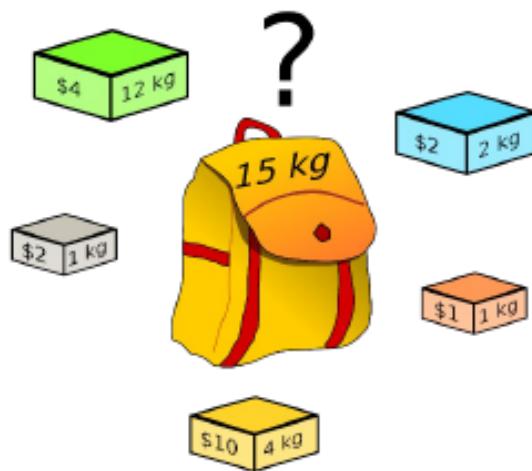


Figura 22.1: problema dal riempimento ottimale

Ottimizzare il riempimento dello zaino:

- ogni oggetto ha un peso e un valore
- determinare il numero di oggetti di ogni tipo in modo tale che il peso sia minore o uguale di un dato limite (valore = capacità dello zaino) e il valore totale sia il maggiore possibile

22.1.2 commesso viaggiatore



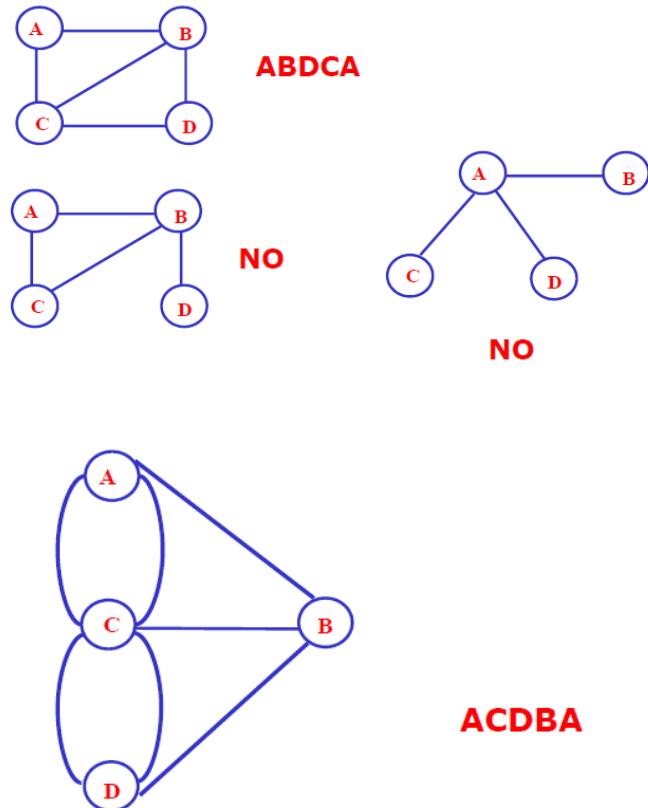
Figura 22.2: trovare il percorso di minore lunghezza che un commesso viaggiatore deve seguire per visitare tutte le città una e una sola volta per poi tornare alla città di partenza

22.1.3 Cammino e ciclo Hamiltoniano

Cammino Hamiltoniano: Dato un multi-grafo, trovare, se esiste, un cammino che tocca tutti i nodi una e una sola volta.

Ciclo Hamiltoniano: Dato un multi-grafo, trovare, se esiste, un ciclo che tocca tutti i nodi una e una sola volta.

Un grafo è Hamiltoniano se possiede un ciclo Hamiltoniano



22.1.4 soddisfattibilità di una formula logica

Definizione 57 SAT: soddisfattibilità di una formula nella logica dei predicati.

Data una formula F con n variabili, trovare, se esiste, una combinazione di valori booleani che, assegnati alle variabili di F, la rendono vera

Es.

$F = (x \text{ and not } x) \text{ or } (y \text{ and not } y)$ $n=2$ non sodd.

$F = (x \text{ and not } y) \text{ or } (\text{not } x \text{ and } y)$ $n=2$ $x=0, y=1$

algoritmi conosciuti oggi per zaino, commesso viaggiatore, ciclo Hamiltoniano e SAT:

provare tutte (o quasi) le combinazioni



complessità esponenziale

Se le variabili che compaiono nella formula sono n , si provano tutte le combinazioni di valori delle variabili, che sono 2^n

22.1.5 Problema del ciclo euleriano

Dato un multi-grafo, trovare, se esiste, un ciclo che percorre tutti gli archi una e una sola volta

Un grafo è Euleriano se possiede un ciclo Euleriano

I ponti di Konigsberg

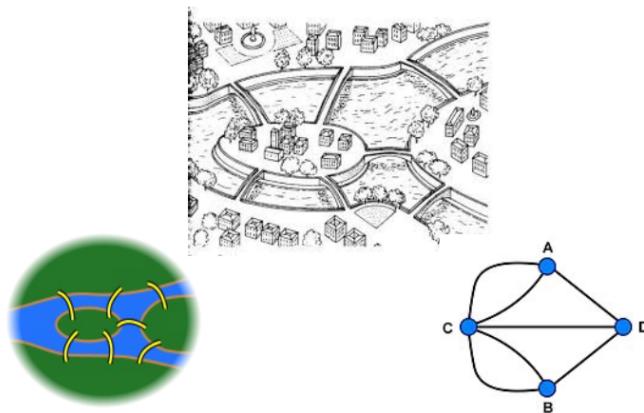
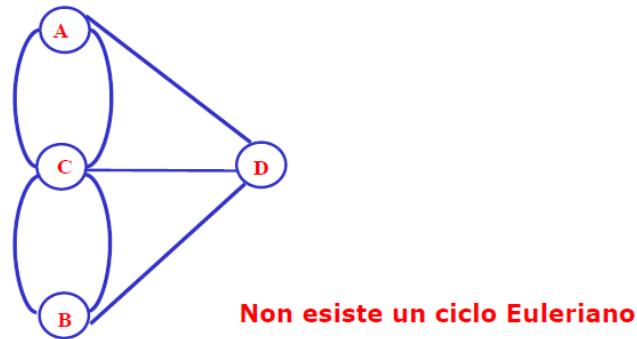


Figura 22.3: passare da tutti i ponti senza passare 2 volte sullo stesso

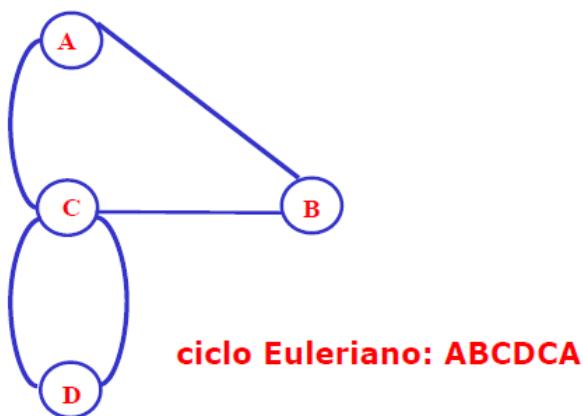
I ponti di Konigsberg: esiste un ciclo Euleriano?



Definizione 58 Teorema di Eulero: Un multi-grafo non orientato contiene un ciclo Euleriano se e solo se gli archi che partono da ogni nodo sono in numero pari.

Quindi basta controllare questa proprietà sul grafo per sapere se un ciclo esiste.

Trovarlo ha complessità polinomiale.



22.2 NP-completezza

Teoria che descrive i livelli di complessità degli algoritmi e ci fa capire dove si collocano questi problemi difficili e se possono essere semplificati.

Definizione 59 Teoria della NP-completezza: .

- Classifica un insieme di problemi difficili
- Si applica a problemi decisionali: con risposta SI o NO

- Ogni problema può essere riformulato come problema decisionale.
- Il problema decisionale ha complessità minore o uguale al problema non decisionale corrispondente. (esiste un ciclo euleriano? vs Trovami un ciclo euleriano)
- Quindi se il problema decisionale è difficile, a maggior ragione lo sarà il corrispondente.

Definizione 60 problemi decisionali: (decisionale è la key-word necessaria nella risposta del quiz riguardo alle domande che parlano di np completezza)

- **Commesso viaggiatore:** dato un intero k, esiste nel grafo un ciclo senza ripetizione di nodi di lunghezza minore di k?
- **Zaino:** dato un valore v, esiste un riempimento dello zaino con valore maggiore o uguale a v?
- **ciclo hamiltoniano:** dato un grafo, esiste un ciclo Hamiltoniano?
- **ciclo euleriano:** dato un grafo, esiste un ciclo Euleriano?
- **Formula logica:** data una formula, esiste un assegnamento alle variabili che rende vera la formula?

22.2.1 Algoritmi nondeterministici

Algoritmi deterministici: (tutti quelli che fino a ora abbiamo visto) scrivo il codice, lo compilo, verifico la sua correttezza... quindi si dice deterministico perché dato un input mi restituisce sempre lo stesso output.

Definizione 61 Algoritmi nondeterministici: (realmente non possono essere implementati)

se io ho più alternative da seguire, questo alg le esegue tutte contemporaneamente e si ferma quando trova la soluzione

Un algoritmo nondeterministico per la soddisfattibilità

```
int nsat(Formula f, int *a, int n) { //a in realtà sono bool
    for (int i=0; i < n; i++)
        a[i]=choice({0,1});
    if (value(f,a))
        return 1;
    else
        return 0;
}
//Restituisce 1 se esiste almeno una scelta con risultato 1
//complessità della scelta quindi anche quella di tutto l'algoritmo
//è O(n)
```

choice mi permette di fare delle scelte all'interno di un insieme (sceglie a caso i valori di a, permette di fare delle scelte)

22.2.2 Un algoritmo nondeterministico di ricerca in array

```
int nsearch(int* a, int n, int x) {
    int i=choice({0...n-1});
    if (a[i]==x)
        return 1;
    else
        return 0;
}
//O(1)
```

Questa choice esplora tutto lo spazio e se l'elemento esiste lo trova (ovviamente si parla sempre di algoritmi non implementabili)

22.2.3 Un algoritmo nondeterministico di ordinamento

```
int nsort(int* a, int n) {
    int b [n];
    for (int i=0; i<n; i++)
        b[i]=a[i];
    for (int i=0; i<n; i++)
        a[i]=b[choice({0..n-1})];
    if (ordinato(a))
        return 1;
    return 0;
}
//O(n)
```

22.2.4 Relazione fra determinismo e nondeterminismo

(per la teoria della np completezza:) Per ogni algoritmo nondeterministico ne esiste uno deterministico che lo simula, esplorando lo spazio delle soluzioni, fino a trovare un successo.

Se le soluzioni sono in numero esponenziale, l'algoritmo deterministico avrà complessità esponenziale.

- **P**= insieme di tutti i problemi decisionali risolubili in tempo Polinomiale con un algoritmo deterministico ($P = \{ \text{ricerca, ordinamento, ciclo Euleriano} \dots \}$)
- **NP**=insieme di tutti i problemi decisionali risolubili in tempo Polinomiale con un algoritmo Nondeterministico ($NP : \text{Nondeterministico Polinomiale}$)

($NP = \{ \text{ricerca, ordinamento, fattorizzazione, soddisfattibilità, zaino, commesso viaggiatore, ciclo Hamiltoniano...} \}$)

(se un problema non appartiene a P non è automatico che appartenga a NP)

NP = insieme di tutti i problemi decisionali che ammettono un algoritmo deterministico polinomiale di verifica di una soluzione (certificato)

Per dimostrare che un problema R appartiene a NP si dimostra che la verifica di una soluzione di R è fatta in tempo polinomiale

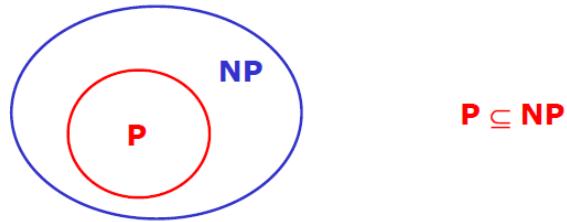
Esempi di verifica

Ciclo Hamiltoniano: si può verificare con complessità $O(n)$ se un cammino è un ciclo Hamiltoniano

Formula logica: si può controllare in tempo $O(n)$ se dato un assegnamento di valori booleani alle variabili rende vera la formula

P = Problemi decisionali facili da risolvere

NP = Problemi decisionali facili da verificare



P = NP ?

Figura 22.4: 1 milione per chi dimostra $P=NP$

22.2.5 Riducibilità

La riducibilità è un metodo per convertire l'istanza di un problema P_1 in un'istanza di un problema P_2 e utilizzare la soluzione di quest'ultimo per ottenere la soluzione di P_1 .

Un problema P_1 si riduce in tempo polinomiale a un problema P_2 se ogni soluzione di P_1 può ottenersi deterministicamente in tempo polinomiale da una soluzione di P_2

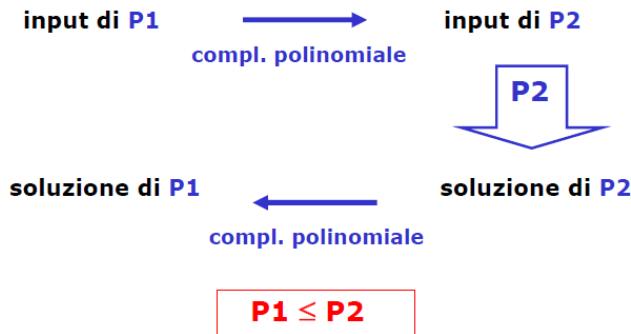
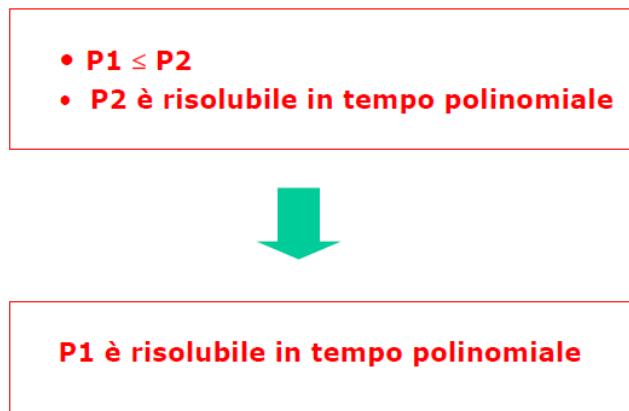


Figura 22.5: (dimostrare che ad esempio il commesso viaggiatore sia risolvibile in tempo polinomiale significherebbe dimostrare $P=NP$)



22.2.6 Teorema di Cook

Definizione 62 Teorema di Cook: Qualsiasi problema R in NP è riducibile al problema della soddisfattibilità della formula logica

$$\forall R \in NP : R \leq SAT$$

Quindi SAT è più difficile di tutti i problemi in NP

22.2.7 NP-completezza

Definizione 63 NP-completezza: un problema R è NP-completo se

- $R \in NP$ e
- $SAT \leq R$ ($= SAT$ si può trasformare in R , SAT deve essere riducibile a quel problema R)

(si va quindi a collegarsi al teorema di Cook)

Se un problema è NP-completo, è difficile tanto quanto SAT e può essere usato al posto di SAT nella dimostrazione di NPcompletezza di un altro problema.

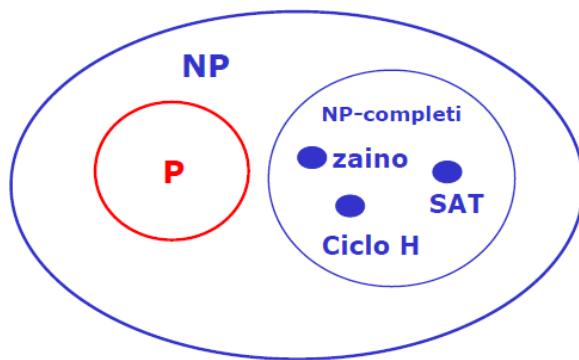
I problemi NP-completi hanno tutti la stessa difficoltà e sono i più difficili della classe NP

Se si trovasse un algoritmo polinomiale per SAT o per qualsiasi altro problema NP-completo, allora tutti i problemi in NP sarebbero risolubili in tempo polinomiale e quindi P sarebbe uguale ad NP

È stato dimostrato che i seguenti problemi decisionali sono NP-completi:

- commesso viaggiatore
- zaino
- ciclo hamiltoniano

moltissimi altri problemi sono stati dimostrati NP-completi



Per dimostrare che un problema R è NP-completo:

- **dimostrare che R appartiene ad NP:** individuare un algoritmo polinomiale nondeterministico per risolvere R (oppure dimostrare che la verifica di una soluzione di R può essere fatta in tempo polinomiale)
- **dimostrare che esiste un problema NP-completo che si riduce a R:** se ne sceglie uno fra i problemi NP-completi noti che sia facilmente riducibile a R

Dimostrare che un problema è NP-completo serve perché non riusciamo a risolverlo con un algoritmo polinomiale e vogliamo dimostrare che non ci si riesce a meno che P non sia uguale ad NP, problema tutt'ora non risolto.

22.2.8 Problema della fattorizzazione di un numero

FATT(Fattorizzazione): scomposizione di un numero in fattori primi
es. $150 = 2 \times 3 \times 5^2$

Come in tutti i problemi di teoria dei numeri, la complessità si calcola in funzione del numero di cifre del numero da fattorizzare (dimensione dell'input)

(bisogna provare tutte le combinazioni quindi è esponenziale)

La moltiplicazione è $O(n^{\log_2 3})$ o $O(n^2)$

```
1.634.733.645.809.253.848.443.133.883.865.090.859.841.783.670.033.0
92.312.181.110.842.389.333.100.104.508.151.212.118.167.511.579 X
1.900.871.281.664.822.113.126.851.573.935.413.975.471.896.789.968.5
15.493.666.638.539.088.027.103.802.104.498.957.191.261.465.571
=
3.107.418.240.490.043.721.350.750.035.888.567.930.037.346.022.842.7
27.545.720.161.948.823.206.440.518.081.504.556.346.829.671.723.286.
782.437.916.272.838.033.415.471.073.108.501.919.548.529.007.337.724
.822.783.525.742.386.454.014.691.736.602.477.652.346.609
```

Richiede meno di un secondo di tempo di calcolo!

Figura 22.6: la moltiplicazione è la verifica della fattorizzazione

L'inverso della moltiplicazione è difficile

Trovare A e B tali che

```
A * B = 3.107.418.240.490.043.721.350.750.035.888.567.930.037.
346.022.842.727.545.720.161.948.823.206.440.518.081.504.556.346.
829.671.723.286.782.437.916.272.838.033.415.471.073.108.501.919.
548.529.007.337.724.822.783.525.742.386.454.014.691.736.602.477.
652.346.609
```

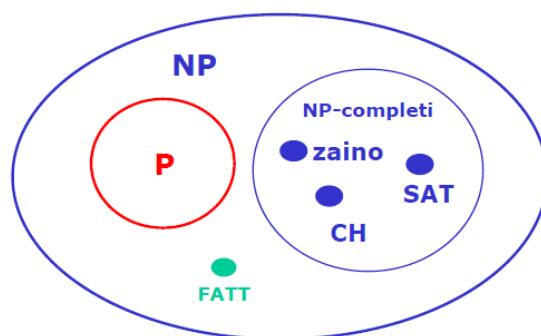
```
A=1.634.733.645.809.253.848.443.133.883.865.090.859.841.783.670
.033.092.312.181.110.842.389.333.100.104.508.151.212.118.167.511
.579
```

```
B= 1.900.871.281.664.822.113.126.851.573.935.413.975.471.
896.789.968.515.493.666.638.539.088.027.103.802.104.498.957.191.
261.465.571
```

Oggi richiede più di una decina di anni di tempo di calcolo!

- Per ora si conoscono soltanto algoritmi esponenziali per FATT
- FATT è in NP: la verifica è polinomiale (moltiplicazione)
- E' questione aperta se FATT sia in P, ma quasi sicuramente non è NP-completo
- Praticamente impossibile scomporre un numero di 200 o più cifre con gli algoritmi attuali

- Sulla difficoltà di FATT si basano i meccanismi della crittografia a chiave pubblica (operazioni facili con inversa difficile) che si basano su numeri primi molto grandi
- Un algoritmo polinomiale per FATT non dimostrerebbe $P=NP$, ma metterebbe in forte crisi i meccanismi della crittografia
- algoritmo polinomiale di Shor (1994) basato su Quantum Computing

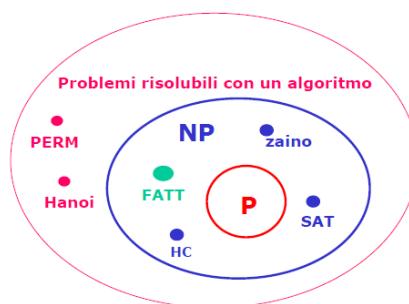


metodologie per affrontare i problemi difficili:

- algoritmi di approssimazione
- algoritmi probabilistici
- reti neurali
- quantum computing

**PERM: Trovare tutte le permutazioni di un insieme
($n!$)**

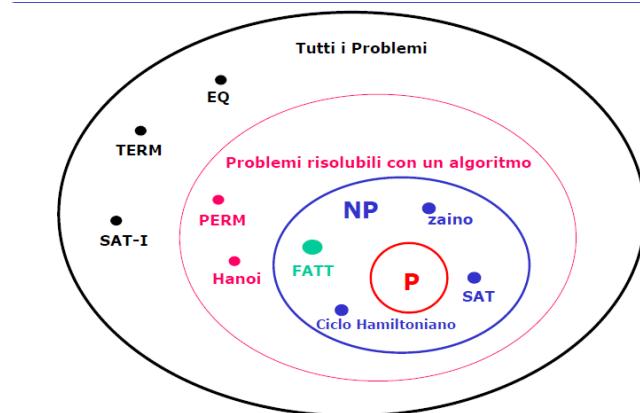
Torre di Hanoi



problemi non risolubili con un algoritmo:

- TERM: decidere la terminazione di un programma su un input

- SAT-I: soddisfacibilità di una formula nella logica del primo ordine
- EQ: decidere l'equivalenza di due programmi



Capitolo 23

Hashing (Lezione 11 Virdis)

23.1 Hashing

(in maniera diretta abbiamo preso l'etichetta dell'oggetto e l'abbiamo usata sia per l'estrazione che per l'indirizzamento ????) ma nel mondo informatico questa situazione non si verifica

La funzione di hash passa da un dominio che può essere anche infinito a un codominio (dominio di output) finito e ben definito

La struttura dati che sceglieremo è quella dell'array in quanto il dominio di output è finito.

23.1.1 Simple hash table (esempio 1)

- trattiamo interi > 0
- chiave coincide con valore
- la funzione hash è la funzione modulo
- convenzione 0 per vuoto

Classe:

```
class HashTable
{
    int * table_;
    int size_; //uguale al modulo della funz hash che stiamo utilizzando
public:
    HashTable( int size );
    bool insert( int key );
    void print();
    int hash( int key );
};
```

Costruttore:

```
HashTable::HashTable( int size )
{
    table_ = new int[size];
    size_ = size;
}
```

per riempire l'array di 0 con complessità $< O(n)$ si deve cercare di ricorrere a funzioni del sistema operativo.

funzione di libreria che si basa su funzione del sistema operativo che a partire da un indirizzo di memoria inserisce un valore per un certo numero di byte ???

```
memset( address , value , size );
```

funzione hash:

```
int HashTable::hash( int key )
{
    return key % size_;
}
```

Insert:

```
bool HashTable::insert( int key )
{
    int index = hash(key);
    if( table_[index] != 0 )
    {
        cout << "already occupied" << endl;
        return false;
    }
    table_[index] = key;
    cout << "key stored" << endl;
    return true;
}
```

23.1.2 collisioni

liste di trabocco

prendo l'array di dimensione size che in ogni casella contiene un insieme di elementi (lista)

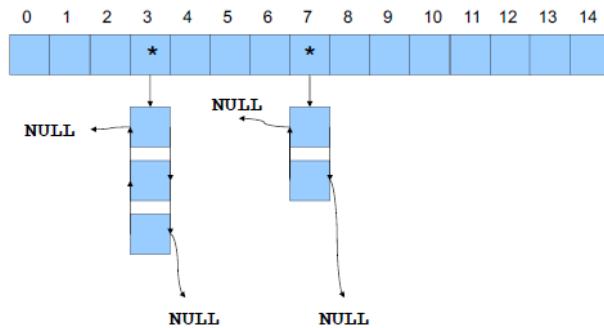


Figura 23.1: si può inoltre implementare la struttura con le liste bidirezionali che ottimizzano la cancellazione (ad esempio) sfruttando una struttura di appoggio (array di puntatori) che contiene gli indirizzi di elementi che possono risultare utili (esempio i maggiori)

```

struct Elem
{
    int key;
    Elem * next;
    Elem * prev;
    Elem(): next(NULL) , prev(NULL) {}
};

class HashTable
{
    Elem ** table_;
    int size_;
    public:
    HashTable( int size );
    bool insert( int key );
    void print();
    void ();
    bool find(int key);
};

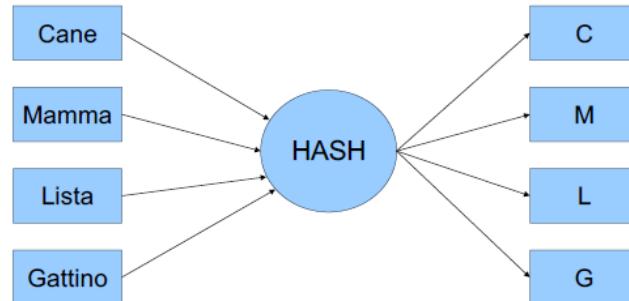
```

open addressing (indirizzamento aperto)

non andando a estendere in senso verticale ma in senso "orizzontale": se la posizione hash è occupata riapplico la funzione hash e trovo il nuovo indirizzo

il problema si presenta in caso di cancellazioni (se poi nella ricerca si trovano elementi vuoti si interrompe la ricerca mentre magari è un buco dovuto a una cancellazione e non la fine di quella serie di elementi di quel determinato hash)

23.1.3 Hashing stringhe



Funzione hash prima lettera

```

int hash(string key)
{
    int index = key[0] % size_;
}
  
```

```

29)
30)
31)
32)
33)
34)
35)
36)
37)
38)
39)
40).....
41).....
42).....
43).....
44).....
45).....
46).....
47).....
48).....
49).
  
```

Figura 23.2: printOccupancy su un articolo di giornale

se nel caso dell'indirizzo 31 l'accesso ha costo costante $O(1)$ nelle liste 42 e 43 ci si avvicina al caso peggiore $O(n)$

Proviamo un'alternativa: somma dei caratteri

```

int hash(){
    for( int i = 0 ; i < key.length() ; ++i )
    {
        index = ( index + key[i] )% size_;
    }
}
  
```

```

27).....
28).....
29).....
30).....
31).....
32).....
33).....
34).....
35).....
36).....
37).....
38).....
39).....
40).....
41).....
42).....
43).....
44).....
45).....
46).....
47).....
48).....
49).....

```

Figura 23.3: stesso articolo di giornale ma con la funzione hash somma dei caratteri: molto migliore del caso precedente

23.2 fine leizone

Quindi quando si ottiene una Good HASH?:

- Dipende fortemente dal tipo di applicazione
- Per applicazioni di indexing e' fondamentale l'uniformità
- Lavorano sulla rappresentazione binaria
- E.g. MurmurHash, CityHash, FarmHash

buone prestazioni si mantengono quando la struttura è grande di circa il doppio degli elementi da considerare (problema quando si ha 1 miliardo di elementi)

23.3 STL map

permette di definire oggetti composti da un valore e una chiave

stl::map

```
std::map < key_T , obj_T > table;
```



```



```

Capitolo 24

Algoritmi di sorting

24.1 selection sort

Definizione 64 selection sort: cerca il minimo portandolo nella prima posizione del vettore, aumenta quindi l'indice minimo e ripete.

```
void exchange( int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
void selectionSort( int A[ ], int n) {
    for (int i=0; i< n-1; i++) {
        int min= i;
        for (int j=i+1; j< n; j++)
            if (A[ j ] < A[min]) min=j;
        exchange(A[i], A[min]);
    }
}
```

24.2 Bubble sort

Definizione 65 Bubble sort: partì da in fondo al vettore e scambi gli elementi "trascinando" il valore più piccolo in cima, poi aumenti l'indice minimo e ripeti.

(si può ovviamente fare anche nell'altro verso con l'elemento più grande).

```
void bubbleSort( int A[], int n) {
    for (int i=0; i < n-1; i++)
        for (int j=n-1; j >= i+1; j--)
            if (A[j] < A[j-1]) exchange(A[j], A[j-1]);
}
```

24.3 insertion sort



Definizione 66 Insertion sort: mano e occhio: confronta, avanzandola ogni volta, la mano con gli elementi precedenti, posizionandola al posto giusto e spostando anch'essi.

```
void sortArray( int arr[] , int len )
{
    int mano = 0;
    int occhio = 0;
    for( int iter = 1 ; iter < len ; ++iter )
    {
        mano = arr[iter];
        occhio = iter-1;
        while( occhio >= 0 && arr[occhio] > mano )
        {
            arr[occhio+1] = arr[occhio];
            --occhio;
        }
        arr[occhio+1] = mano;
    }
}
```

24.4 Quick sort



Definizione 67 Quick sort: partendo da un perno (per esempio l'elemento centrale dell'array per comodità) si prendono due cursori che scorrono fino a trovare a sx un elemento che non è minore del perno e a dx un elemento che non è maggiore del perno per poi scambiarli. Questo finché i due cursori non si incontrano e chiamare ricorsivamente la funzione sui sue sottovarri prima e dopo il perno.

```
void quickSort(int A[], int inf=0, int sup=n-1) {
    int perno = A[(inf + sup) / 2], s = inf, d = sup;
    while (s < d) {
        while (A[s] < perno) s++;
        while (A[d] > perno) d--;
        if (s > d) break;
        exchange(A[s], A[d]);
        s++;
        d--;
    };
    if (inf < d)
        quickSort(A, inf, d);
    if (s < sup)
        quickSort(A, s, sup);
}
```

continua a 8.2

24.5 Merge sort



Definizione 68 Merge sort: dividi a metà l'array sfruttando la ricorsione fino ad avere array di 1 solo elemento per poi attraverso la combina ogni volta unire gli array riempiendoli con gli elementi inseriti in ordine.

```
void combina( int* arr, int start, int mid, int end){

    int iSx=start, iDx = mid;
    vector<int> tempResult;
    while(true){
        if(iSx == mid){
            while(iDx <= end)
                tempResult.push_back(arr[iDx++]);
            break;
        }
        if(iDx > end){
            while(iSx < mid)
                tempResult.push_back(arr[iSx++]);
            break;
        }
        if( arr[iSx] < arr[iDx]){
            tempResult.push_back(arr[iSx++]);
        }
        else {
            tempResult.push_back(arr[iDx++]);
        }
    }
    for(int i = start, j = 0; i <= end; i++, j++){
        arr[i]=tempResult[j];
    }
}

void mergeSort(int * arr, int start, int end){
    int mid;
    if(start < end) {
        mid = (start + end)/2;
        mergeSort( arr, start , mid);
        mergeSort( arr, mid + 1, end);
        combina(arr, start, mid + 1, end);
    }
}
```

24.6 STL sort

sort (first, last);
overloading: sort (first, last, comparatore);
(comparatore è una funzione booleana)

24.7 HeapSort

14.2 17.1.6



24.8 CountingSort

16.3.1

24.9 RadixSort

16.3.2