Prog 2 Orale Appunti di Simone Scionti

Static:

La parola riservata static ha diversi utilizzi.
Generalmente, ha a che fare con l'allocazione statica, infatti ogni variabile dichiarata con l'utilizzo della parola riservata static, viene allocata nel data segment, ovvero quella sezione di ram dedicata all'allocazione statica.
Una variabile allocata staticamente è visibile solo ed esclusivamente dal documento nel quale è dichiarata.

Osservazione:
Se viene usata nella dichiarazione di una variabile o di una funzione globale, essendo che una variabile dichiarata globalmente viene già allocata nel data segment come variabile statica, l'unica funzione che ha è quella di **rendere non visibile** la variabile o la funzione ***da altri documenti.***

Utilizzi della parola riservata static:

Funzioni:
Se viene dichiarata una variabile locale di una funzione, con l'utilizzo della parola static, essa diventa una variabile globale, che viene quindi allocata e deallocata al di fuori del ciclo di vita della funzione. Questo permette di conservare i suo contenuto nel tempo, anche dopo la terminazione dell'esecuzione della funzione.

Classi:
Tutti i membri statici di una classe vengono allocati all'esecuzione del programma e deallocati alla terminazione. Non sono quindi relazionati alla vita di alcun'istanza della classe stessa.
Static var:
La parola riservata static, se usata in una variabile membro di una classe, fa assumere a quella variabile il ruolo di var condivisa tra tutte le istanze della classe stessa.
(come se fosse una variabile globale per tutti gli oggetti di quella classe).
Static method:
Un metodo membro di una classe statico, è un metodo che può essere invocato senza alcun oggetto chiamante, usando ad esempio il risolutore di scope "::" , si dice infatti che il metodo appartiene alla classe, e non alle istanze della classe. Ovviamente un metodo statico non può accedere ad alcun attributo di alcuna istanza della classe, non essendo appunto legato a nessun oggetto.

Register:

La parola riservata Register ci permette di dichiarare variabili che, si spera, verranno allocate nei registri del processore(è una richiesta). Questo permetterà un notevole incremento della velocità di accesso al dato memorizzato dalla variabile, in quanto non si passa per il bus, ma si accede direttamente ai registri.

Ovviamente non si può risalire all'indirizzo di una variabile allocata in un registro del processore.
Queste variabili sono spesso utili come contatori nei cicli, e **non possono essere globali.**

Extern:

Lo specificatore di ambito extern, ci viene in aiuto quando vogliamo ampliare lo scope di una variabile globale, dal documento in cui è dichiarata a tutti gli altri documenti sorgenti di un programma. Infatti basta utilizzare la parola riservata extern seguita dal tipo e dal nome della variabile, per renderla leggibile nel documento sorgente in cui ci troviamo.
Osservazione:
Di default tutte le funzioni globali sono extern, visibili da tutti i documenti di un programma.

Binding

Il Binding, nell'ambito della programmazione ad oggetti, è il processo che permette di associare un metodo alla sua implementazione. Ci sono due tipi di Binding: Dinamico e statico.

Statico: il compilatore prepara una sorta di mappa a tempo di compilazione, riguardo le associazioni tra i metodi e le loro implementazioni.

Dinamico:
Il binding dinamico permette di associare la chiamata di un metodo ad una sua implementazione, a run-time. Questo è necessario ad esempio nel caso del polimorfismo, e quindi dei metodi virtuali. Sarà in base al tipo dell'oggetto a runtime che si sceglierà che implementazione dello stesso metodo associare alla chiamata.

- Allocazione dinamica e statica, heap e stack

Data segment:
La sezione Data segment della RAM, è la sezione nella quale vengono allocate le variabili staticamente. E' la sezione gestita più velocemente, e si può "forzare" l'allocazione in questa sezione di ram tramite l'utlizzo della parola riservata static.
Heap:
L'heap è quella sezione della memoria centrale , nella quale viene allocato spazio dinamicamente, ovvero in fase di runtime, nel caso del c++ con gli operatori **new** e **delete.**
La gestione dell'heap è più lenta rispetto alla gestione dell'allocazione statica.

Stack: (LIFO)
Lo stack è invece quella sezione di memoria utilizzata principalmente per salvare i record di attivazione, e quindi ci permette di invocare funzioni, per poi tornare una volta terminata l'esecuzione della funzione, alla istruzione successiva rispetto alla chiamata della funzione. Lo stack ci permette quindi pure di poter operare ricorsivamente.

Inoltre, una volta definita una funzione, tutte le sue variabili locali vengono allocate nello stack. Questo da vita a quella che definiamo “memoria automatica”.

-Paradigma divide et impera

Il paradigma divide et impera si suddivide in 3 fasi

1. **Decomposizione:**
   Decomporre i dati in sottoinsiemi più semplici
2. **Ricosrsione:**
   Applicare la risoluzione su problemi semplici grazie ai dati decomposti

3. **Ricombinazione:**
   Ricombinare le soluzioni ottenute per ottenere la soluzione al problema iniziale

-Meccanismo di espansione per rendere un array “dinamico”

La maggior limitazione degli array, è quella di dover specificare a priori un limite di spazio riservato per la struttura stessa. Generalmente si tende a specificare uno spazio leggermente più grande di quello che si è consapevoli di dover utilizzare , ma ci sono volte nelle quali non è possibile nemmeno fare una stima, e di conseguenza si dovrebbe ricorrere all’uso di strutture dinamiche come le liste ad esempio, piuttosto che usare gli array.
Sappiamo che le liste e le strutture dinamiche in generale hanno dei vantaggi rispetto agli array, ma in realtà ci sono anche degli svantaggi, infatti le strutture dinamiche permettono solo un accesso sequenziale, mentre gli array permettono tramite l’utilizzo dell’indice, l’accesso diretto.

Qual’è allora il modo di usare gli array, ma “quasi” senza dover prefissare una dimensione, decidendola in un certo senso a run time?

Bisogna ricorrere a dei meccanismi di espansione, sicuramente costosi in termini di tempo, ma necessari.
Di fatto, questi meccanismi implicano il fatto che bisogna, nei casi in cui si deve allargare l’array, costruire un array più grande e copiare il contenuto del vecchio array in quello nuovo. La prima cosa che viene spontaneo pensare è che nel momento in cui viene occupata l’ultima posizione libera dell’array, si crea un array con una posizione in più e si esegue il procedimento di copia. Certamente questo sarebbe un meccanismo molto lento, perchè la copia degli elementi verrebbe eseguita ad ogni inserimento in una qualunque posizione superiore al limite dell’array iniziale.

L’idea:
In realtà il modo meno costoso e più funzionale è quello di raddoppiare la dimensione dell’array, ogni volta che si arriva ad occupare l’ultima casella.

In questo modo, **il costo di gestione dell'array aumenta in maniera logaritmica** rispetto al numero di elementi, in quanto (*supponendo che la copia degli elementi venga fatta in tempo quasi costante con il memcpy()*) e supponendo di partire da un array di un solo elemento, al primo inserimento viene subito raddoppiata la sua dimensione, e quindi creato un array secondario di dimensione doppia rispetto a quello di prima, e copiato il suo contenuto. Successivamente, questa operazione viene rieseguita quando l'array arriva ad essere occupato in 4 caselle, poi in 8, 16 e così via dicendo. **All'aumentare degli elementi questa operazione viene eseguita sempre meno volte.**
E' anche vero però che all'aumentare degli elementi potremmo ricadere in un maggiore spreco di spazio, ma questo è un compromesso spazio-tempo da dover accettare.

Overheading di funzioni(sovraccaricamento)

L'hoverheading o overloading di funzioni, consiste nel dichiarare e definire una funzione con lo stesso nome di una già dichiarata e definita, ma con un diverso numero di parametri o con almeno un parametro diverso per quanto riguarda il tipo.

Overriding di un metodo

L'overriding è una tecnica che ci permette di ridefinire il comportamento di un metodo di una classe, in una sua sottoclasse. In questo caso ovviamente il metodo sarà lo stesso, con lo stesso numero di parametri e gli stessi parametri.

Complessità - lez1

complessità di un algoritmo:
Per "misurare" la complessità di un algoritmo, dobbiamo avvalerci di strumenti di calcolo che prescindano dall'architettura su cui gira quell'algoritmo, in maniera tale da poter dare una stima della complessità computazionale dell'algoritmo stesso.
Per esprimere la complessità asintotica di un algoritmo nel caso peggiore di esecuzione, ovvero quello con il maggior numero di passi elementari, si usa la notazione O().
Generalmente, nel calcolo della complessità di un algoritmo si cerca di evidenziare quante volte vengono visitati gli elementi. Questo è spesso riconoscibile ad esempio attraverso il numero di cicli annidati , o comunque spesso intuitivamente.
Ad esempio gli algoritmi di ordinamento basilari, hanno generalmente una complessità di O(n^2), in quanto dovendo eseguire il confronto di ogni elemento con tutti gli altri, necessitano di due cicli annidati.

appunti:
dipende da vari parametri, infatti possiamo vedere un algoritmo come un unica funzione che sia lineare o meno. lineare se dipende da un parametro costante, che rende proporzionale(rispetto al numero di elementi che utilizza al suo interno) il numero di operazioni.

UN algoritmo con complessità di tipo logaritmica, all'aumentare del numero di elementi su cui lavora, non aumenta la sua complessità proporzionalmente, ma la aumenta sempre meno.

Ricorsione - lez2

In informatica viene detto **algoritmo ricorsivo** un algoritmo espresso in termini di se stesso, ovvero in cui l'esecuzione dell'algoritmo su un insieme di dati comporta la semplificazione o suddivisione dell'insieme di dati e l'applicazione dello stesso algoritmo agli insiemi di dati semplificati.

Tale tecnica risulta particolarmente utile per eseguire dei compiti ripetitivi su di un set di dati. L'algoritmo richiama se stesso generando una sequenza di chiamate che ha termine al verificarsi di una condizione particolare che viene chiamata **condizione di terminazione o caso base**, che in genere si raggiunge con particolari valori di *input*.

La tecnica ricorsiva permette di scrivere algoritmi eleganti e sintetici per molti tipi di problemi comuni, anche se non sempre le soluzioni ricorsive sono le più efficienti. Questo è dovuto al fatto che comunemente la ricorsione viene implementata utilizzando le funzioni, e che l'invocazione di una funzione ha un costo rilevante(punti di esecuzione salvati sullo stack per poi tornare lì una volta terminata l'esecuzione della funzione chiamata, e tutte le successive chiamate che ne conseguono). Questo rende più efficienti gli algoritmi iterativi.

la ricorsione funziona tramite il così detto "albero di ricorsione", ovvero, prima viene chiamata più volte la stessa funzione, si arriva al caso base, e poi in cascata si risale trovando la soluzione per ogni funzione chiamata precedentemente.

esempi:
in realtà si tratta di una lista di chiamate nel caso del fact, ma ad esempio nel caso della successione di fibonacci, ad ogni chiamata della funzione, vengono fatte due chiamate ricorsive. in quel caso infatti si tratta precisamente di un albero, con più casi base, nelle foglie sotto.

Algoritmi di ordinamento -lez3

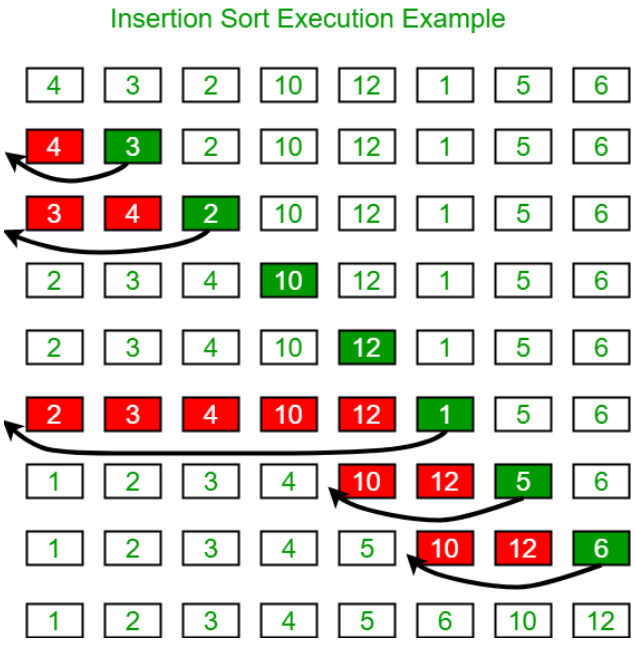
Nome	Migliore	Medio	Peggiore	Memoria	Stabile	*In place*
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$\Theta(1)$	No	Sì
Insertion sort	$O(n)$	$O(n+d)$	$O(n^2)$	$\Theta(1)$	Sì	Sì
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$\Theta(1)$	Sì	Sì
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$	No	Sì

osservazione sulla stabilità:
è importante fare in modo che oltre ad ordinare per il valore degli elementi, si tenga conto anche della posizione iniziale in cui si trovava l'elemento nell'array, un pò come se fosse una coda, in cui ogni elemento ha un suo valore(che rappresenta la categoria di priorità) ma una sua personale priorità data dalla posizione in cui si trova inizialmente nell'array. un algoritmo che rispetta queste due cose è detto algoritmo stabile di ordinamento.
Ad esempio se si ordina una sequenza di studenti per anno di corso, supponendo che gli studenti fossero già ordinati alfabeticamente, un algoritmo di ordinamento che produce una sequenza di studenti ordinati per anno di corso, ma mantenendo l'ordine alfabetico in ognuno di questi anni di corso, è detto stabile.

Osservazioni sul limite inferiore di complessità:
Un teorema dice che qualsiasi algoritmo di ordinamento non può mai scendere sotto O(n Log n).
Osservazione "In place":
Un algoritmo di ordinamento si dice "in place" quando per raggiungere il risultato finale non ha bisogno di array d'appoggio.
Nel caso del quick sort, viene utilizzato altro spazio per le chiamate ricorsive, ma si definisce in place in quanto non vengono salvati gli elementi dell'array su appoggi ausiliari.
Selection sort:
Preso un array di n elementi, il Selection sort agisce così:
Si cerca il più piccolo degli elementi **in tutto l'array** e si mette al posto 0. A questo punto al prossimo colpo di ciclo, si cercherà il più piccolo degli elementi **a partire dalla posizione 1**, e si metterà in posizione 1. Così via dicendo.

```
for(int i = 0; i<n;i++){
Int posMin = i;
        for(int j = i+1; j<n;j++){ //la j parte sempre da i+1 perchè le posizioni <i sono
                                                state ordinate nei colpi di ciclo precedenti, e il
                                                posMin è uguale ad i, l'elemento in cui
                                                andremmo a posizionare il minimo.
        if(array[j] < array[posMin] ) posMin = j;
        }
        Int tmp = array[i]
        array[i] = array[posMin];
        array[posMin] = tmp;
}
```

Insertion Sort



È un algoritmo tipicamente iterativo, semplice da implementare.
Si usano due indici, ad ogni iterazione il primo indice(j) punta all' elemento iesimo, dove i è il contatore del ciclo, e l'altro indice (k) all'elemento successivo. Se il secondo elemento(k) è minore del primo(j), allora viene fatto un ciclo al contrario, che continua a spostare il primo indice(j). Il ciclo finisce quando l'elemento puntato dal primo indice (j) è minore dell'elemento puntato dal secondo indice (k). A questo punto l'elemento puntato da K , viene inserito nella posizione j, e gli elementi da j a k-1 vengono shiftati di un elemento, occupando le posizioni da j+1 a k, in quanto la posizione j sarà occupata dall'elemento che è stato spostato.
Eseguendo questo procedimento per ogni elemento si ottiene l'array ordinato.

Naive selection //da cercare

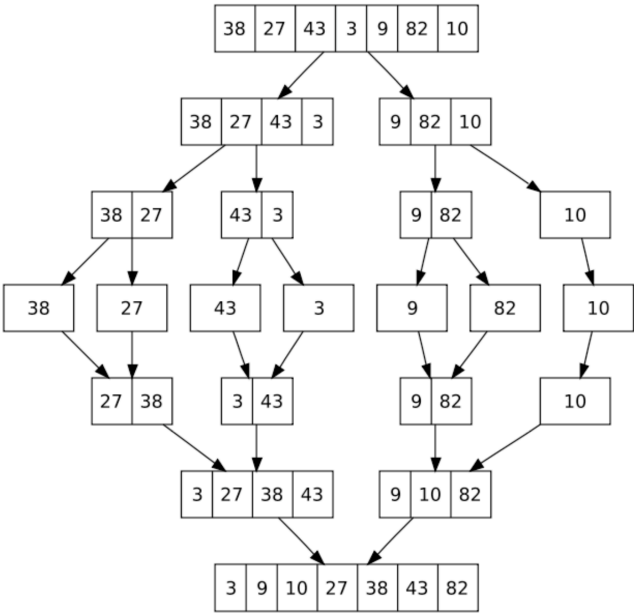
un algoritmo che riesce a selezionare l'iesimo elemento (in maniera ordinata), ma senza avere l'array ordinato, ordinando solo delle partizioni di array che ci interessano, ragionando in binario.

Merge sort:

Il **merge sort** è un algoritmo di ordinamento basato su confronti che utilizza un processo di risoluzione ricorsivo, sfruttando la tecnica del Divide et Impera, che consiste nella suddivisione del problema in sottoproblemi della stessa natura di dimensione via via più piccola.

Opera in questo modo:
Fase 1. partizionamento
Preso un array di n elementi, si prende l'elemento medio, e si divide l'array in due sottoArray.
Fase 2. fusione
Una volta ordinati questi due sottoArray, basterà fonderli tra loro ovviamente in maniera ordinata.
(Questo set di operazioni viene ripetuto su ogni sottoArray di riferimento, fino a raggiungere per ogni sotto array il caso base nel quale l'array è composto da un elemento solo. Sarà da lì che si procederà alla fusione e risalendo nell'albero

ricorsivo chiudendo le chiamate, si fonderanno tutti gli altri array fino a raggiungere la chiusura della prima chiamata aperta e quindi il risultato finale.)




```
void mergesort(Item a[], int
left, int right) {
if (left<right) { //se left ==
right --> num elementi = 0;
 int center = (left+right)/2;
 mergesort(a, left,center);
 mergesort(a,center+1,right);
 merge(a, left, center, right);
 }
}
```

Ad ogni chiamata, la fusione dei due array verrà eseguita sulla base che i due array sono già ordinati, e quindi la procedura Merge funziona in questo modo:

```
Void merge(int *a, int start, int middle, int end){
    middle +=1; //l'elemento di mezzo verrà visitato tramite lo start,
che lo comprende
    int i = 0; //usato per l'array di appoggio
    while(start <= middle && middle <= end){
        if( a[start] < app[middle] ) {app[i] = a[start]; start++;}
        else{ app[i] = a[middle]; middle++;}
        i++;
    }
    //adesso bisogna controllare per quale condizione si è usciti dal
ciclo. Se si è usciti per la seconda condizione, vorrà dire che start
non è arrivato a middle, in quanto gli elementi selezionati da start
erano sempre maggiori di quelli dell'array di destra. Allora:

    if(start<= middle) //siamo usciti perchè middle è arrivato ad end,
tutto l'array di destra è stato copiato nel posto giusto.
        while (i <= end) {app[i] = a[start]; start++;}

    else  //siamo usciti perchè start è arrivato a middle, l'array di
sinistra è stato copiato al posto giusto.
        while(i<= end) {app[i] = a[middle]; middle++;}

//adesso basta copiare il contenuto dell'array app, su a.Lo si può fare
con memcpy
    memcpy(a,&app,sizeOf(a[0])*end+1);
}
```

Quicksort

il quicksort funziona in maniera similare al mergesort, solo che piuttosto che lavorare tanto sulla fusione degli array(ordinando durante la fusione), lavora ordinando durante la divisione(**partizionamento**), e la fusione viene addirittura eseguita in tempo costante, inoltre a differenza del mergesort è un algoritmo di ordinamento in place.

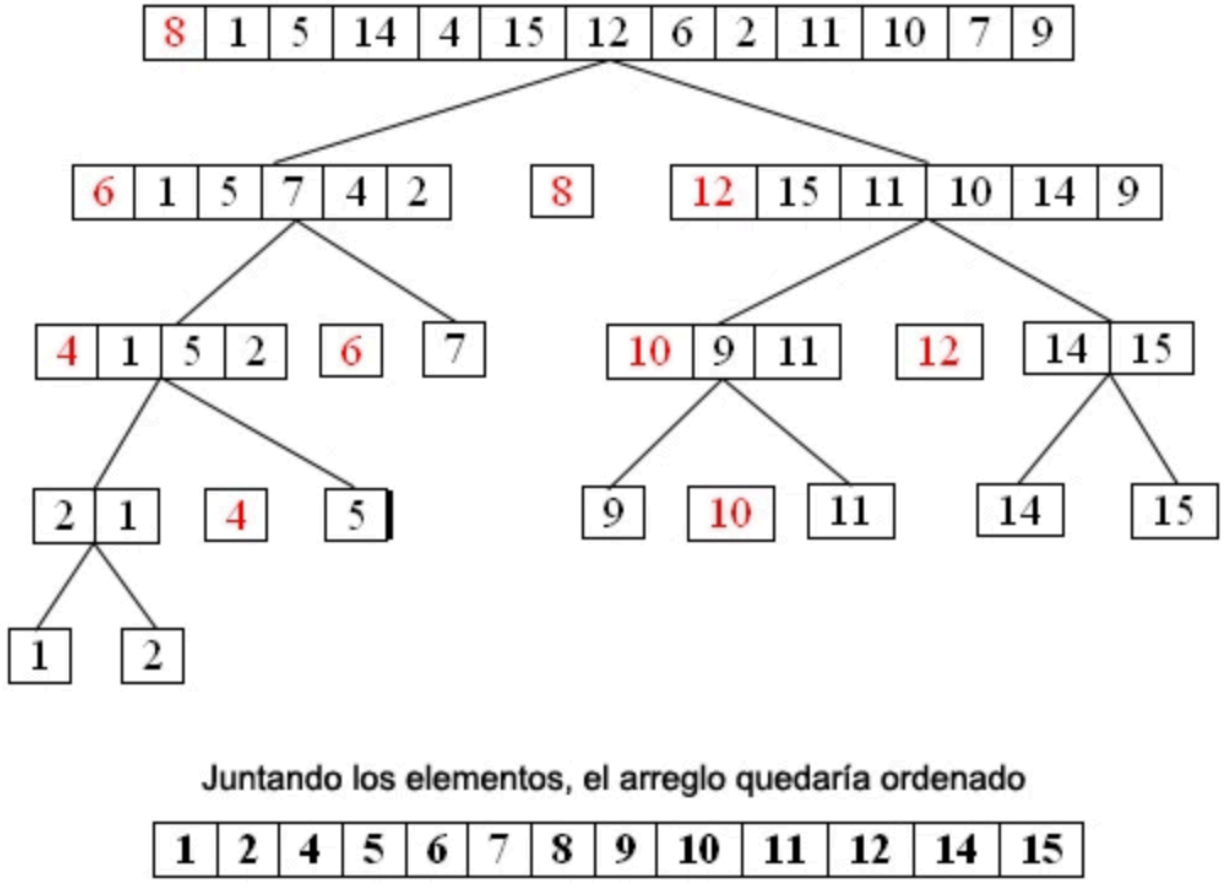
il quicksort, prende l'array, e esegue ricorsivamente questa procedura: spacca a metà l'array, (scegliendo il pivot di mezzo in maniera casuale, o quasi) fa in modo che nell'array di sinistra ci siano tutti elementi più piccoli del pivot, e in quello di destra tutti più grandi del pivot.

Questa cosa viene fatta utilizzando due indici, uno che parte dalla posizione 0 e uno che parte dalla posizione n.
Confrontando gli elementi in posizione i e j,
A questo punto (per ognuno dei due elementi) se l'elemento rispetta la condizione di appartenenza a quella metà dell'array, permette all'indice che lo punta di essere incrementato.
Se l'elemento invece non rispetta la condizione, allora se anche l'altro elemento dei due non la rispetta , vengono scambiati i due elementi e si incrementano entrambi gli indici, sennò se è uno solo dei due e non rispettare la condizione, l'altro indice viene incrementato fin quando non punterà ad un elemento che anch'esso non rispetta la condizione, in modo che possa avvenire lo scambio.
Questo procedimento termina quando i > j.

sintetizzando:
se a[i] < pivot && a[j] >= pivot —> i++, j++ (entrambi rispettano la condizione)
Se a[i] >= pivot && a[j] >= pivot —> j++ (a[i] NON rispetta la condizione < pivot)
Se a[i] < pivot && a[j] < pivot —> j++ (a[j] NON rispetta la condizione >= pivot)
Se a[i] >= pivot && a[j] < pivot -> SWAP(i , j); i++, j++ (entrambi NON rispettano la condizione, quindi swap)

- dopo ciò viene ri-eseguito lo stesso procedimento sia sull'array a destra del pivot, che su quello a sinistra.
- L'esecuzione della fase di **partition** termina nel momento in cui si hanno tanti piccoli array di due elementi(o uno) (questi sono i casi base) e spaccandoli a metà, non fa altro che mettere l'elemento più piccolo a sinistra.

//in fine quindi la fusione di tutti questi piccoli array risulta essere eseguita in tempo costante perchè non si deve fare alcun lavoro di ordinamento, ma soltanto unirli. (*questi piccoli array risultano essere le foglie e i pivot rimasti , se rappresentiamo l'algoritmo con un corrispondente albero)*




```
void quickSort(vector<int>& A, int low,int high)
{

          /*r risulterà essere il pivot scelto, che sarà
          quindi usato per stabilire dove finisce il primo ed
          inizia il secondo array delle prossime esecuzioni*/

          int r=partition(A, low,high);
          quickSort(A,low,r);
          quickSort(A,r+1,high);

}
```

il problema sta nel riuscire a spezzare l'array e calcolare un numero medio in maniera tale da poter confrontare con esso gli elementi per capire se devono andare a sinistra o a destra.
Più siamo fortunati nella scelta del pivot, più l'algoritmo viene eseguito velocemente.

La scelta del pivot(ad ogni passaggio) influenza notevolmente la velocità dell'algoritmo, più si riesce a costruire due array di uguale misura, a partire da un singolo array ad ogni passaggio, più l'algoritmo è veloce. Caso medio(O(n log n)).

Nel caso peggiore, ovvero quello in cui si ha uno dei due array costituito da un solo elemento, ad ogni passaggio, aumenta notevolmente l'altezza dell'albero di ricorsione, e questo porta ad usare più spazio nello stack, ed a rendere più lento l'algoritmo.

Generalmente si sceglie come pivot il primo elemento dell'array, ma questo rende l'algoritmo Lentissimo nel caso in cui gli venga dato un array già ordinato, paradossalmente.
Al di là di questo, è consigliabile scegliere il pivot invocando una funzione random. È dimostrabile che la probabilità di scegliere un pivot che faccia ricadere nel caso peggiore è vicina allo zero.

NB. Il numero massimo di chiamate, nel caso peggiore del quicksort è n-1. Quindi il numero di chiamate ricorsive cresce proporzionalmente ad n. Questo dipende dalla scelta del pivot, o accade se l'array è già ordinato.

Liste - lez7

è possibile implementare una lista, usando però gli array, al fine di sfruttare i vantaggi riguardanti la complessità delle operazioni di inserimento ed estrazione, che, una volta fissata la posizione in cui eseguire l'operazione, si eseguono in tempo costante O(2).

questa cosa può essere fatta in due modi :

1) uso un array come contenitore di dati , ed un altro array come contenitore di indici. Il secondo, contiene degli indici dei successivi. quindi se abbiamo due array a1{1,4,2} e a2{2,-1,1}. il successivo del numero 1 è il numero che trovo in posizione 2....

2) uso un solo array, nel quale nelle posizioni pari inserisco gli elementi che voglio conservare, mentre nelle posizioni dispari gli indici. quindi il successivo di ogni elemento è dettato dall'indice salvato nella cella successiva, che sarà una cella dispari.

Liste dinamiche lez-8

le liste **dinamiche** funzionano diversamente rispetto alle precedenti. infatti nelle precedenti rimaneva un problema legato agli array, ovvero la dimensione prefissata a priori.
Si può ovviare a questo problema ricorrendo a meccanismi di espansione, oppure implementando una lista DINAMICA.

inoltre il vantaggio è che usiamo lo spazio strettamente necessario, ad ogni elemento che serve lo allochiamo al momento, dinamicamente. Non presupponiamo alcuna dimensione.
lo spazio per salvare il puntatore è proporzionale al numero di elementi, e questo ci basta e avanza.

Lista doppiamente linkata:
lista doppiamente linkata o doppiamente concatenata: è una lista in cui ogni nodo ha due puntatori, un next e un prev, in maniera tale da poter scorrere la lista in entrambi i versi.Ovviamente c'è un link da sistemare in più, ad ogni operazione di inserimento e cancellazione.

Template

I template son un importante strumento che ci permette di definire il comportamento di una classe o di una funzione, in funzione di un tipo di dato generico, permettendo quindi il riutilizzo del codice, per diversi tipi di dato.
Come sappiamo ci sono operazioni fattibili in maniera del tutto equivalente con qualsiasi tipo di dato : ad esempio le operazioni di inserimento e di cancellazione all'interno di una lista, che vengano fatte con un numero intero o con un tipo di dato composto che descrive uno studente, possono essere eseguite allo stesso modo.
Ci sono invece operazioni che devono essere eseguite diversamente in funzione del tipo di dato sul quale vengono eseguite(solitamente i confronti). In quel caso bisogna definire l'overloading degli operatori utilizzati.

NB. In realtà la clausola template, non fa altro che duplicare automaticamente il codice, creando quindi le corrispondenti funzioni o classi per il tipo di dato che serve.

Liste circolari - lez 10

Una lista circolare è in poche parole una lista nella quale non c'è ne una testa ne una coda. In realtà , se partissimo da una normale lista, il nodo di testa e di coda andrebbero semplicemente concatenati tramite il puntatore next, per ottenere una lista circolare.

Bisogna mantenere un puntatore current, cioè al nodo attuale, in quanto non essendoci dei puntatori di riferimento sempre validi come la tail o la head,

operazioni come l'inserimento vanno eseguite a partire dal nodo puntato da current. *Ovviamente questo non nega la possibiltà di eseguire ad esempio un inserimento ordinato, basterebbe scorrere tutta la lista fin quando non si trova il punto giusto in cui inserire, ma lo si fa a partire dal current.*

In sostanza quindi , in una lista circolare, scorrendola attraverso i puntatori, non capiterà di gestire il caso in cui il puntatore punta a NULL, escluso il caso particolare della lista vuota.

Alberi - lez11

Gli alberi sono una struttura dati di natura ricorsiva, **che possiamo definire come una lista in cui ogni nodo può avere più di un successore.**
É di natura ricorsiva perchè lo possiamo vedere come un nodo legato a k alberi(con k numero di figli per nodo), oppure come l'insieme vuoto(def. ricorsiva). Quindi ogni nodo che non è una foglia avrà un sottoalbero.
Nelle liste , a partire da un nodo possiamo identificare un successore ed un predecessore, mentre nell'albero è più corretto chiamare il predecessore padre ed i successori figli.
Inoltre sappiamo che in una lista l'unico nodo senza successore è la coda, e l'unico nodo senza predecessori è la testa.
Nel caso dell'albero l'unico nodo senza padre è la radice, e gli unici nodi senza figli sono le foglie(ovvero i nodi finali delle diverse diramazioni), che ovviamente potranno essere più di una.

Un qualsiasi nodo che non sia una foglia è detto nodo interno all'albero, esclusa la radice.

Una importante caratteristica degli alberi è che per arrivare ad ogni nodo a partire dalla radice o da un altro nodo, c'è un solo percorso utilizzabile, e (nel secondo caso) lo si decide cercando l'antenato (predecessore) comune più vicino e poi seguendo il percorso per arrivare al nodo desiderato.

-altra caratteristica è che a partire da ogni nodo, esiste una "lista di predecessori" per arrivare alla radice.

Definizioni:

Discendenti di un nodo:

- Dato un nodo, i nodi che appartengono al suo sottoalbero si dicono **Discendenti.**

Ascendenti di un nodo:

- Dato un nodo, i nodi per cui si passa percorrendo il cammino dalla radice al nodo in questione, vengono detti **Ascendenti.**

Nodi fratelli:
Due nodi sono detti fratelli se discendono dallo stesso padre.

Livello di un nodo:
Il livello di un nodo è la sua distanza dalla radice.
Due nodi che sono fratelli si trovano sicuramente allo stesso livello, ma non tutti i nodi dello stesso livello sono fratelli.

Profondità o altezza dell'albero:
Si può definire ricorsivamente una funzione di calcolo in quanto la radice ha profondità 0, mentre i figli della radice profondità 1, e cosi via fino alle foglie.
Alla fine l'altezza risulta essere la profondità del nodo con maggiore profondità, il livello maggiore.

NEL CASO DI UN ALBERO BINARIO DI RICERCA:

```
int _altezza(Nodo<H> *x){
   if(!x) return 0;
   return max(_altezza(x->getRight()),_altezza(x>getLeft())) + 1;
 }
```

Albero bilanciato o equilibrato:
Un albero si dice bilanciato o equilibrato se : dato k numero di figli per nodo, ogni nodo interno all'albero(compresa la radice) ha esattamente k figli.

Limiti sui nodi (distinzione tra i diversi tipi di alberi)

I diversi tipi di alberi, si differenziano principalmente grazie ad i limiti imposti sui nodi.
NB. Il rango o grado di un nodo è il numero di figli del nodo.
Se imponiamo un limite di ***un figlio per ogni nodo*** *quindi* ***rango 1***, un albero diventa una lista(anche detta albero unario).
se ogni nodo dell'albero ha due figli **(rango 2)** è detto albero binario.

Un'altra distinzione tra tipi di alberi può essere fatta in base al fatto che l'albero sia posizionale o meno. **Un albero si dice posizionale se i figli di ogni nodo sono ordinati secondo un ordinamento prefissato.**

Albero binario

Definizione ricorsiva albero binario ->

1) nodo collegato a due alberi

{

2) albero vuoto

le **configurazioni di un nodo** in un **albero binario posizionale sono 4** :
figlio a destra, figlio a sinistra, due figli sia a destra che sinistra, nessun figlio.

le **configurazioni di un nodo** in un **albero binario NON posizionale sono 3**: figlio sotto, due figli sia a destra che a sinistra, nessun figlio.

Osservazioni sull'albero binario:

Ad ogni livello, un albero binario può contenere al massimo 2^n nodi.
Il numero massimo di nodi di un albero di altezza n, è [2^(n+1)]-1.

Metodi di visita Alberi binari :

Generalmente la visita di un albero può essere fatta in due modi:

Profondità:
Partendo dalla radice, dopo il nodo corrente viene esaminato uno dei suoi figli. Se non ci sono figli si ritorna al padre del nodo corrente (backtracking semplice) per poi andare verso un'altro figlio e così via. Accade che tutti i discendenti di ogni nodo vengono visitati prima dei nodi di pari livello.

Ampiezza:
Partendo dalla radice, dopo il nodo corrente, vengono visitati tutti i nodi di pari livello, per poi passare a quelli di livello successivo.

Albero binario di ricerca

BST = Albero binario di ricerca **(binary search three).**
L'albero binario di ricerca è un **albero binario posizionale ordinato**, in cui indichiamo il **primo figlio di un nodo a sinistra**, e **il secondo a destra.** Questo ci permette di ordinare.

caratteristica di un **albero binario posizionale ordinato**, è che presi 3 nodi, di cui uno padre e due figli è che : il padre contiene (nell'ordinamento utilizzato) un valore compreso tra quello del figlio a sinistra e quello del figlio a destra
(sx < padre < dx)->(insieme a sx < padre < insieme a dx).

osservazione:
esiste un solo modo di inserire in maniera corretta, rispettando le dovute condizioni, i valori all'interno di un albero(premesso che la struttura dell'albero sia fissata).
Se non è fissata una struttura grafica, in base all'ordine con cui si inseriscono(magari con **l'algoritmo di inserimento sotto descritto**), si ottengono alberi diversi, e quello da preferire è sempre l'albero con altezza minore, in quanto le operazioni da eseguire diventano meno complesse.

Per possibile domanda all'orale**(Natural Fill)**
- preso quindi un insieme di numeri, e immaginando la struttura dati bisogna contare , la radice di quanti nodi deve essere più grande. si ordinano i numeri in linea, e si sceglie quindi l'elemento iesimo, dove i corrisponde al numero di nodi dei quali la radice deve essere più grande.
- **ripetendo questa cosa per ogni sotto albero si possono inserire gli elementi.**

Ricerca di massimo e minimo

per trovare l'elemento più piccolo di un albero basta andare sempre verso sinistra fin quando non si sfora dall'albero

per trovare il più grande sempre verso destra fin quando non si sfora

Ricerca di succ e prec

per trovare il successore (a livello di valore ordinato) di un nodo si fa un passo sotto a destra e poi sempre a sinistra(il successore è il più piccolo dei maggiori)

per trovare il predecessore di un nodo si fa un passo a sinistra e poi sempre a destra(il più grande dei minori)

Osservazione:
C'è un caso in cui questa cosa non funziona, ovvero quando si cerca il successore di un nodo che non ha figli.
in tal caso si esegue un altro algoritmo:

successore:
Si sale verso la radice tramite i padri, e appena faccio il primo passo **a destra** quello è il successore.(potrebbe anche essere il primo passo).
predecessore:

Si sale verso la radice tramite i padri, e appena faccio il primo passo **a sinistra** quello è il predecessore(potrebbe anche essere il primo passo).

```
Nodo<H> * _succ(Nodo<H> * tmp){
        if(tmp->getRight() == NULL){
            Nodo<H> *p = tmp->getParent();
            while(p != NULL && p->getKey() <= tmp->getKey()) p = p->getParent();
            return p;
        }
        else{
            return _min(tmp->getRight());
        }
```

Metodi di visita pre-order , in-order e post-order:

pre order:
visito in ordine-> padre, figlio sx, figlio dx (ovviamente con concezione ricorsiva dei sottoalberi come figli)

post order:
visito in ordine-> figlio sx, figlio dx, padre.

in order:
visito in ordine -> figlio sx, padre, figlio dx.

Notare che “pre” -“post”-“in” sono in riferimento al padre.

Operazioni di inserimento cancellazione e ricerca in un albero binario

Inserimento :
un buon algoritmo per l'inserimento su un albero già esistente(o con un solo elemento) è quello di cercare quell'elemento all'interno dell'albero, e appena si sfora(NULL) si inserisce l'elemento lì.
osservazione: quindi quando inserisco un nuovo nodo, esso sarà sempre una foglia dell’albero.

```
BST<H>* insert(H x){
        Nodo<H> * node = root;
        Nodo<H> *p = NULL;
        while(node != NULL){
            p = node;
            if(x>node->getKey()) node =  node->getRight();
            else node = node->getLeft();
        }
        Nodo<H> * newn = new Nodo<H>(x);
        n++;
        if(p == NULL){
            root = newn;
```

```
            return this;
        }
        if(x<p->getKey()) p-> setLeft((newn));
        else p->setRight((newn));
        newn->setParent(p);
        return this;
    }
```

Cancellazione:
E' la procedura più complicata delle 3.
Ovviamente la cancellazione di un nodo NON deve portare alla conseguente cancellazione dei suoi sottoalberi.
Quindi ci sono tre casi da distinguere :

1. **Il nodo non ha figli :**
   in questo caso la procedura è banale, basta cancellare il nodo e settare il puntatore del padre a NULL.
2. **Il nodo ha un solo figlio:**
   In questo caso basterà prendere il figlio e metterlo al posto del nodo(con tutto il suo conseguente sottoalbero).
3. **Il nodo ha due figli:**
   in questo caso si eseguono questi passi:
   Si cerca il successore del nodo X che vogliamo eliminare e:

   Se il successore è figlio di X allora basterà sostituire il valore del successore con quello di X, ed eliminare il nodo originariamente successore.

   Se il successore non è figlio di X(ovviamente sta comunque nel suo sottoalbero di sinistra) allora bisognerà sostituire il valore del successore Y con quello di X, e poi sostituire al posto di Y il suo sottoalbero destro.

   (il figlio sx di Y sarà sempre == NULL, perchè è il successore , e quindi è il più piccolo del sottoalbero di destra, un passo a destra e poi si prende l'ultimo andando a sinistra)

   Questo si fa settando come figlio sx del padre di Y, il figlio destro di Y(creando un ponticello).

Ricerca:

```
Nodo<H> * _search(H x){
       Nodo<H> * tmp = root;
       while(tmp && tmp->getKey() != x){
           if(x < tmp->getKey()) tmp = tmp->getLeft();
           else tmp = tmp->getRight();
       }
       return tmp;
   }
```

COMPLESSITA' operazioni base in un albero:

la complessità di search, insert, minimo, massimo, succ, canc è pari all'altezza dell'albero nel caso peggiore(O(h)).

quindi la complessità delle operazioni dipende da come è composto l'albero, quando l'albero non è bilanciato la complessità risulta essere O(n).

-Quindi se si è fortunati, e si inseriscono le chiavi nel giusto ordine, l'albero che viene fuori è bilanciato, e permette di avere una complessità minore nelle operazioni
nel caso migliore, l'altezza di un albero(quindi che risulti bilanciato) è log(n)
Nel caso peggiore, l'altezza dell'albero è esattamente n.
//come si fa a mantenere bilanciato l'albero?

Grafi -lez13

Un grafo è un insieme di vertici (o nodi) e archi. Gli archi risultano essere delle coppie di vertici, ordinate se il grafo è direzionale, non ordinate se non lo è.

Un grafo è una struttura che risulta essere una generalizzazione degli alberi. Infatti nel caso degli alberi, ogni nodo ha un solo punto di entrata , e k punti di uscita con altri possibili k nodi collegati.
Ne caso di un grafo, possono esserci per ogni nodo punti di entrata multipli e punti di uscita multipli.
Nei grafi non è definito un ordinamento , come invece lo era nel caso di un albero binario di ricerca.

Un grafo può essere sia direzionale che non, e questo dipende dal fatto che la relazione (tra gli elementi contenuti in esso) sia simmetrica o meno.
se è simmetrica-> non direzionale. *(v1 collegato a v2 e v2 collegato a v1)*
asimmetrica->direzionale.(diventano quindi coppie di elementi ordinati).

DEFINIZIONE DI CAMMINO

<v1,v2> è un cammino se (v1,v2) appartiene all'insieme delle coppie che identificano gli elementi in relazione tra loro, insieme detto "**E**".
<v1,v2,v3> è un cammino se (v1,v2) e (v2,v3) appartengono a **E.**

<v1, v2, v3, … , vn> è un cammino se(V_1,V_2) ,(V_2,V_3),…,(V_{n-1}, V_n) appartengono ad E.

DEFINIZIONE DI CICLO

un ciclo è un cammino <v1,v2....vn> tale che v1 = vn.
un **ciclo semplice** è un ciclo in cui un arco non è mai ripetuto più volte.

GRAFO ACICLICO

E' un grafo che non contiene cicli

GRAFO CONNESSO

Un grafo si dice connesso se e solo se per qualsiasi coppia (Vi,Vj) si prende, esiste un cammino che va dall'uno all'altro elemento : <Vi…Vj>
proprietà : un grafo connesso orientato, ha sempre più archi di un grafo connesso non orientato.(non direzionale)

TORNANDO ALLA DEFINIZIONE DI ALBERO

Un albero risulta essere un grafo non orientato , connesso e aciclico.

Rappresentazione di un grafo

Per quanto riguarda l'implementazione di un grafo, in realtà viene implementato costruendo una classe che abbia un semplice array, ed una delle due cose seguenti, matrice di adiacenza o liste di adiacenza.

Ogni elemento contenuto nel principale array, sarà un vertice del grafo, mentre la matrice di adiacenza, o le liste di adiacenza, vengono utilizzate per definire i collegamenti tra i diversi vertici.
Un grafo può essere rappresentato quindi in due modi :

Tramite matrice di adiacenza:

la matrice di adiacenza è una matrice composta solo da 1 e 0. la presenza di un 1 in corrispondenza degli indici i,j indica che l'iesimo nodo del grafo è collegato al j-esimo nodo. lo 0 indica il fatto che i due nodi non sono collegati.

Osservazione:

la rappresentazione con matrice di adiacenza di un grafo non orientato implica il fatto che la matrice sia simmetrica rispetto alla diagonale

principale, in quanto ogni coppia (i,j) presuppone che esista anche la coppia (j,i) in quanto il grafo non è orientato.

Tramite liste di adiacenza:

ogni nodo del grafo ha una lista nella quale sono contenuti tutti i nodi adiacenti al nodo in questione.
Se quindi, preso un nodo X, nella lista di adiacenza di X non è contenuto il nodo Y, Y NON è adiacente ad X.

Osservazione:

Questa rappresentazione è conveniente in termini di spazio, in quanto a differenza della matrice di adiacenza, dove bisogna indicare la non adiacenza con la presenza di uno 0 nella casella corrispondente, in questo caso si ha la non presenza di un determinato nodo nella lista di adiacenza di un altro nodo.

VISITA DI UN GRAFO

VISITA BFS (breadth-first search)

la BFS è una visita in ampiezza: prende come parametro un nodo sorgente appartenente ai vertici, dal quale parte la visita.

- scelto un nodo sorgente da cui parte la visita, viene suddiviso il grafo in sezioni di distanza dalla sorgente(distanza 1= tutti i nodi adiacenti alla sorgente e cosi' via.)

osservazione:
questo algoritmo è quindi in grado di capire la distanza di ogni nodo dalla sorgente(quindi può essere utile per calcolare questa distanza)

un nodo a distanza k viene visitato sempre prima di un nodo a distanza k+1.

per questo algoritmo si usa ***un vettore delle visite,*** utile a farci capire quando un nodo è già stato visitato.
se ci interessa sapere anche la distanza dalla sorgente di ogni nodo(numero di archi percorso per arrivarci) si implementa un altro ***vettore distanze.***
se ci interessa sapere i ***percorsi minimi*** di ogni nodo dalla sorgente, si usa un altro ***vettore(parent)*** che contiene il padre di ogni nodo.
l'unico nodo che non avrà un genitore è la sorgente.

inoltre si usa una coda per inserire tutti i nodi adiacenti alla sorgente(inizialmente), e ad ogni iterazione viene poi preso uno degli elementi della coda e viene controllato se esistono adiacenti che non sono ancora stati inseriti in coda. se è così allora viene tolto quel nodo, e vengono aggiunti in coda i nodi adiacenti, e viene ripetuto lo stesso ragionamento. tutto ciò fin quando la coda diventa vuota(abbiamo già inserito tutti gli elementi una volta.)

```
void BFS(H x){
        int i =  k->findIndex(x);
        if(i<0) return;
```

```
        initAllVectors(); // inizializziamo tutti i vettori che usiamo
        Queue<int> q; // coda che contiene gli indici sui quali lavoriamo
        q.enqueue(i);
        visited[i] = 1;
        distances[i] = 0;
        while(q.size() != 0){
            int v = q.dequeue();
            for(int u = 0; u< k->size(); u++){// inserisco gli adj
                if(adj[v][u] == 1) // u adiacente a v
                    if(!visited[u]){
                        q.enqueue(u);
                        visited[u] = 1;
                        distances[u] = distances[v] +1;
                        parents[u] = v;
                    }
            }
        }
    }
```

VISITA DFS(depth first search)
La visita DFS è una visita in profondità
si cerca di partire dalla sorgente e percorrere il cammino più lungo possibile, passando per i nodi una volta sola(fin quando non si incontrano solo nodi nei quali si è già passati). appena si incontrano solo nodi già visitati, si torna indietro di un nodo, e si fa lo stesso tentativo.
in questo tipo di visita, ogni nodo può trovarsi in tre stati diversi durante la visita.
bianco = non visitato
grigio = visita iniziata (a partire da quel nodo).
nero = visita terminata(ho visitato tutti i possibili percorsi a partire da quel nodo).

OSSERVAZIONE:
si può implementare una informazione riguardante un "**tempo**", cioè sostanzialmente degli intervalli che identificano il momento di apertura e di chiusura di ogni chiamata ricorsiva (il tempo viene incrementato ad ogni azione della DFS). in questa maniera è possibile stabilire poi, se esiste un cammino tra un nodo ed un altro, perchè in tal caso , l'intervallo di apertura e chiusura dell'uno sarà compreso in quello dell'altro.

COMPONENTI CONNESSE
una componente connessa è un sottoinsieme del grafo (dell'insieme dei vertici) massimale(più grande possibile) i cui nodi godono della proprietà di **mutua raggiungibilità.** Sono delle classi di nodi nelle quali prendendo una coppia di nodi(A,B), esiste un cammino da A->B e da B->A.
osservazione:
Questo può accadere solo se ci sono cicli, altrimenti le componenti connesse sono i nodi stessi, e non esiste un sottoinsieme nel quale si presenta la condizione di mutua raggiungibilità.

COMPONENTI FORTEMENTE CONNESSE
Stessa cosa ma in un grafo orientato.

Per trovare i sottoinsieme di componenti connesse, basta guardare i cicli nel grafo. Trovati n cicli, e specificati i nodi interni ai cicli, se ci sono nodi in comune tra un ciclo e l'altro, vengono concatenati, in quanto rappresentano un unica componente connessa.
Questo è l'algoritmo di "tarjan" inventato negli anni 70-80 (usa una visita bfs)

Esiste un altro algoritmo più intuitivo (detto a***lgoritmo per la ricerca delle componenti connesse***) che usa due visite DFS:

1. ***visita il grafo con una visita DFS*** (complessità asintotica della DFS)
(per calcolare i tempi di fine visita F(i) per qualsiasi i da 0 a n (num elementi)
Questa DFS calcolerà più alberi DFS, a partire da radici differenti magari.
osservazione:
l'unico caso in cui viene visitato tutto l'albero a partire da una sola radice, è il caso nel quale tutto il grafo è una componente connessa, ovvero esiste un cammino da qualsiasi nodo a qualsiasi altro nodo.

2. ***Calcola il grafo trasposto rispetto a quello di partenza***(il grafo con le direzioni opposte, si fa facendo la trasposta della matrice di adiacenza)(non ha senso eseguire questo passo su un grafo non direzionale)

3.*visita il grafo G^t appena ottenuto con una nuova DFS.*
Questa seconda esecuzione ci permette di isolare le componenti connesse dal resto del grafo
(utilizzando i tempi di fine visita già calcolati dalla precedente DFS)
(invece di partire dal nodo 0, parte dal nodo con tempo di fine visita maggiore e andando a scendere.)
alla fine le componenti connesse sono quelle individuate dalle stesse radici che abbiamo trovato, in quanto da ogni radice parte un ciclo non abbinabile ad un altro ciclo del grafo.
quindi se abbiamo trovato un albero a partire da una radice, una componente connessa è quell'albero, dalla radice alla fine della visita DFS.

4. ***restituisce gli alberi DFS che sono calcolati al punto 3 come se fossero le diverse componenti connesse del grafo.***

La complessità di questo algoritmo è
O(n)[inizializzazione] + O(n)[]
la dfs visit viene invocata ricorsivamente da ogni nodo, una ed una sola volta. (dopo l'invocazione della dfs pubblica generale).
ognuna di queste dfs invocate da ogni nodo impiega O(n) per esaminare gli archi adiacenti, dalla matrice di adiacenza. la complessità totale della dfs visit con matrice di adiacenza è è quindi O(n^2). Questo se implementato con una lista di adiacenza, mi permetterebbe di impiegare meno tempo, in quanto controllerei solo

ed esclusivamente gli adiacenti al nodo dal quale parte la visita. In tal caso, in totale verrà impiegato tempo O(n+m) dove m è il numero di archi totali nel grafo. (visito una volta tutti i nodi, e una volta tutti gli archi)(meglio di così non si può fare).

ORDINAMENTO TOPOLOGICO

Se si ha un grafo in cui non ci sono cicli, è possibile fare un ordinamento topologico.

Se , una volta eseguito l'algoritmo per le componenti connesse, si rappresentasse ogni componente connessa come un singolo nodo, essendo che tra le diverse componenti connesse non ci sono cicli, si potrebbe eseguire un ordinamento topologico.

UN grafo connesso ha una sola componente connessa

un grafo aciclico ha tante componenti connesse quanti i nodi, ogni nodo è una componente connessa

Calcolo di potenze esempio ottimizzazione Complessità

```
int power(int x, int n){ //complessità O(n)
    if(n == 0) return 1;
    return x * power(x, n-1);
}


int power(int x, int n){ //complessità O(log n)
    if(n == 0) return 1;
    if(n%2 == 0) {
        int y = power(x,(n)/2);
        return y * y;
    }
    if(n%2 == 1){
        int y = power(x,(n-1)/2);
        return x * y * y;
    }
}
```