GLOSSARIO

- **STABILITA =** un algoritmo è detto stabile se preserva l'ordine tra record con la medesima chiave
- **LOCALITA** = Un algoritmo è detto in loco se non necessita di strutture ausiliarie
- **STRUTTURA DATI =** struttura che specifica l'organizzazione delle informazioni che permette di realizzare e implementare un determinato tipo di dati
    - **LISTE CONCATENATE =**Insieme di nodi collegati linearmente in modo consecutivo. Ogni nodo viene definito dal dato che deve rappresentare e un riferimento al nodo successivo.
    - **STACK =** Collezione di dati organizzati in LIFO
    - **CODE =** Collezione di dati organizzati in FIFO
    - **ALBERO** = Struttura dati a nodi che associa ad ogni nodo due successori detti figli. Un albero è un grafo non orientato connesso e privo di cicli
        - **ALBERO PERFETTAMENTE BILANCIATO =** un albero è perfettamente bilanciato quando pr ogni nodo la differenza in valore assoluto tra i numeri di nodi presenti nei suoi sottoalberi destro e sinistro è al massimo 1
        - **ALBERO BILANCIATO** = un albero è detto bilanciato quando per ogni nodo la differenza in valore assoluto tra le altezze dei suoi sottoalberi sinistro e destro è al massimo 1
        - **ALBERO QUASI COMPLETO** = un albero è quasi completo quando è completo almeno fino al penultimo livello ovvero ogni nodo di profondità minore di h-1 possiede entrambi i figli.
- **GRAFO =** Un grafo è un insieme di vertici ed archi e possono essere non orientati oppure orientati
    - **CAMMINO =**
    - **CAMMINO SEMPLICE =** cammino che non presenta vertici ripetuti
    - **CICLO =** cammino da vertice x a vertice x
    - **CICLO SEMPLICE =** ciclo nel quale si ripete solo il vertice iniziale alla fine
    - **CATENA =** serie finita di archi consecutivi, che si connettono
    - **CIRCUITO** = Catena con vertice iniziale uguale al vertice finale
    - **CIRCUITO HAMILTONIANO** = circuito che passa per ogni vertice del grafo una e una sola volta (grafi NON orientati)
    - **CIRCUITO EULERIANO** = circuito che attraversa ogni nodo una e una sola volta
    - **GRAFO CONNESSO =** Grafo in cui tra ogni coppia di vertici esiste una catena
    - **GRAFO FORTEMENTE CONNESSO =** grafo in cui tra ogni coppia di vertici esiste un cammino
    - **CRICCA =** sottografo completo
    - **FORESTA =** insieme di alberi
    - **ALBERO RICOPRENTE MINIMO =** albero ricoprente di un grafo pesato avente la minor somma di pesi

TECNICA DIVIDE ET IMPERA

- Nella tecnica divide et impera si ha un problema P da risolvere e poi una istanza del problema P.
- Tecnica che consiste nel dividere un problema in sottoproblemi piu piccoli risolvibili in tempo minore del problema di base e una volta scomposti in sottoproblemi e ottenute le soluzioni andiamo a ricomporle nel problema di base. Questi sottoproblemi devono possedere un caso base che definisce la terminazione della scomposizione in sottoproblemi
- Un caso in cui divide et impera funziona bene sono mergesort e quicksort. Il mergesort in termini di memoria sfrutta una quantità logaritmica di chiamate ricorsive. IL quicksort nel caso migliore usa lo stesso spazio ma nel caso peggiore va a creare un albero della ricorsione di altezza n e va ad occupare uno spazio eccessivo. Nel caso peggiore l'operazione di divide del quicksort andrebbe a scomporre l'array in due porzioni, una di altezza 1 e una di altezza n-1. Questo è un caso molto raro e quindi statisticamente non accade spesso. Nel caso medio il quicksort è ottimo quindi si utilizza molto.

TECNICA GREEDY

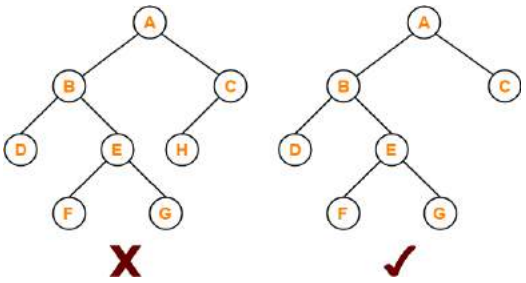
- Questa tecnica in una sequenza di passi a partire dall'insieme vuoto costruisce una soluzione ammissibile
- Ad ogni passo espande una soluzione parziale già ottenuta e poi termina quando non è piu possibile espandere la soluzione parziale
- Tra le possibili scelte di espansione, la soluzione deve avere alcune caratteristiche
    - La soluzione parziale deve soddisfare i vincoli del problema.
    - Si sceglie la soluzione che al momento ci appare migliore.
    - la scelta è irrevocabile.

TECNICA PROGRAMMAZIONE DINAMICA

- **Differenza con la tecnica divide et impera:**
    - con la tecnica divide et impera si parte dalla istanza del problema e la si scompone in sotto istanze risolte in modo ricorsivo. Si parla di tecnica TOP DOWN
    - la prog. dinamica è bottom up. Utilizza una struyttura dati che mantiene le soluzioni ai problemi. Parte dal caso base e man mano risolve problemi piu complessi fino arrivare al problema iniziale
    - nel divide et impera la soluzione dei problemi procedono in parallelo. Di conseguenza se risolviamo in una istanza parallela un problema che ci serve in un'altra istanza, in quell'altra istanza dobbiamo comunque risolverlo a prescindere dal fatto che lo abbiamo già risolto parallelamente in un'altra istanza. (esempio Fibonacci). La programmazione dinamica essendo bottom up ci permette di sfruttare le soluzioni ottenute per alcuni problemi per risolvere problemi lentamente piu complessi.

STRUTTURA DATI HEAP

- Uno heap è un albero binario **quasi completo** in cui la chiave contenuta in ciascun figlio è o maggiore o uguale oppure (quindi max-heap) minore o uguale (quindi min-heap) delle chiavi contenute nei figli.
- Si considerano per comodità heap in cui le foglie dell'ultimo livello si trovano piu a sinistra possibile
- Un albero binario è quasi completo quando è completo almeno fino al penulitmo livello.
    - **MAX** num di nodi: 2^{h+1}
    - **MIN** num di nodi: 2^{h}
    - **Altezza** h = $\log_2 n$



- La radice degli heap è il nodo con il valore maggiore
- Le operazioni di modifica vanno spesso a danneggiare la struttura dello heap e quindi ad ogni inserimento bisogna effettuare una procedura di **risistemazione dello heap**:
    - Si prende l'ultima foglia e il suo contenuto si mette nella radice
    - Se la nuova radice non è al suo posto (ovvero è piu piccola di entrambi o uno dei suoi figli)allora devo metterla al suo posto
    - Guardo quindi tra i suoi figli chi devo fare salire per ottenere uno heap (quindi prendo il figlio maggiore) e lo faccio salire
    - Faccio scendere la radice al posto del nodo figlio che ho fatto salire e continuo dal punto 2

- La procedura risistema richiede circa θ(altezza) confronti
- Per costruire uno heap ci sono due tecniche:
    - **TOP-DOWN**: tramite divide et impera (utilizza memoria aggiuntiva):
        - Se un albero è vuoto allora è uno heap (caso base)
        - Se un albero non è vuoto allora:
            - Si chiama ricorsivamente la procedura creaHeap su albero sinistro
            - Si chiama ricorsivamente la procedura creaHeap su albero destro
            - Si risistema l'albero con la procedura definita prima

    - **BOTTOM-UP**: dai sottoalberi piu piccolo a quelli piu grandi
        - È prevista una lettura dello heap in orizzontale da destra a partire dalle foglie
        - Una foglia è uno heap e non c'è niente da fare
        - Si sale di un livello e si analizza sempre da destra a sinistra
        - Per ogni nodo che osservo applico la chiamata risistema passando come parametro il suo sottoalbero

CODA CON PRIORITA'

- Collezione di dati da cui gli elementi vengono prelevati seocndo un <u>criterio di priorità</u>
- Ogni elemento ha una chiave, chiave piu bassa ➔priorità maggiore
- Implementa operazioni di:
    - findMin() → θ (1)
    - deleteMin()→ θ (logn)
    - insert(elemento e, chiave k) → θ (logn)
    - delete(elemento e) → θ (logn)
    - changeKey(elemento e, chiave d) → θ (logn)
- vengono implementate mediante **MinHeap**

UNION FIND

- permette di rappresentare una collezione di insiemi disgiunti tramite le operazioni di:
  - **UNION(A,B):** unisce gli insiemi A e B in un unico insieme di nome A
  - **FIND(x)**: restituisce il nome dell'insieme che contiene l'elemento x
  - **MAKESET(x):** crea un nuovo insieme {x} di nome x
- Una partizione è una foresta di alberi
- Ogni insieme è rappresentato da un albero con radice:
  - Nodi: elementi dell'insieme
  - Radice: nome dell'insieme
- ALGORITMI **QUICK FIND**:
  - Algoritmi in cui FIND è l'operazione privilegiata
  - Gli elementi dell'insieme sono foglie
  - Il nome dell'insieme è inserito nella radice
  - MAKESET è un'operazione facile visto che basta costruire un albero con una radice e un figlio aventi lo stesso valore, richiede O(1)
  - FIND è anch'essa un'operazione immediata poiché basta guardare il puntatore alla radice dell'elemento e si ottiene subito il nome della radice. Richiede O(1)
  - UNION è un pochetto piu complessa poiché devo modificare i puntatori di B (che voglio unire ad A), il tempo quindi è pari ad O(n). Una cosa intelligente da fare è, se voglio unire A e B e A ha meno elementi di B, modifico i puntatori di A e non quelli di B. Il costo rimane sempre O(n). tramite una analisi ammortizzata si vede che dopo n operazioni di makeset e O(n) operazioni UNION e FIND mediamente otteniamo un tempo di O(logn).
- ALGORITMI **QUICKUNION**
  - Elementi dell'insieme sono all'interno dei nodi
  - Elemento che da il nome all'insieme si trova all'interno della radice.
  - Si utilizzano alberi di diversa altezza
  - MAKESET: crea semplicemente un nodo con il valore al suo interno e quindi richiede tempo O(1)
  - UNION: richiede tempo O(1) poiché semplicemente lego l'albero B tramite puntatore al nodo A.
  - FIND : Nel caso peggiore mi chiede O(n) poiché devo risalire al massimo tutta l'altezza dell'albero per arrivare alla radice . Per migliorare questo basta semplicemente evitare che l'albero cresca troppo. Se devo fare una operazione di UNION(A,B) quindi vado ad unire all'albero di altezza maggiore quello di altezza minore. Nel caso io unisca A a B in questo caso poi devo anche scambiare le due radici per manetnere la caratteristica del nome (ovvero che facendo UNION(A,B) la radice principale dell'unione è A e non B nonostante io unisca A a B)
  - ogni albero quickunion costruito effettuando 0operazioni di UNION bilanciate contiene almeno $2^{rank(x)}$ nodi, con x radice ➔ **#nodi ≥ $2^{rank(x)}$**
  - è possibile effettuare un'operazione di compressione del c ammino che riduce ulteriormente il tempo del FIND. Se effettuo una operazione di FIND all'inizio io devo scorrere fino alla radice per sapere il nome del suo insieme. Nel farlo io percorro un cammino e tutte le radici che incontro sono alla fine figlie di quell'insieme, quindi posso staccare le radici dalle radici a cui sono connesse e collegarle direttamente alla radice principale per ridurre le future operazioni di FIND
  - Con questa analisi ammortizzata ottengo che FIND ha un costo di $O(\log^* n)$, ovvero un costo iterato che quindi dipende dal numero di operazioni che effettuo

OPERAZIONE	MAKESET	UNION	FIND
Quickfind	O(1)	O(n)	O(1)
QuickFind bilanciata	O(1)	O(logn) ammortizzata	O(1)
QuickUnion	O(1)	O(1)	O(n)
QuickUnion bilanciata	O(1)	O(1)	O(logn)

ALBERI BINARI DI RICERCA

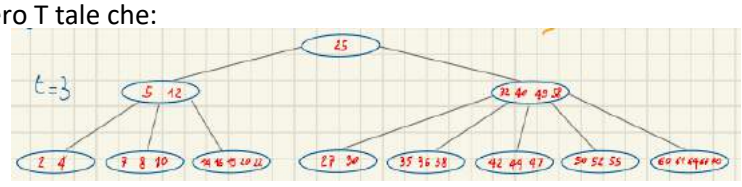
- Un albero binario di ricerca è un albero binario in cui per ogni nodo n
    - Il valore di ogni chiave contenuta nel sottoalbero **sinistro** di n è **minore** della chiave contenuta in n
    - Il valore di ogni chiave contenuta nel sottoalbero **destro** di n è **maggiore** della chiave contenuta in n
- Il numero massimo di nodi di un albero di altezza h è 2^{h+1}-1
- Un albero è detto **bilanciato** quando per ogni nodo la differenza in valore assoluto tra le altezze dei suoi sottoalberi sinistro e destro è al massimo 1
- Alberi di Fibonacci di altezza h = albero AVL con il numero minimo di nodi
- Gli inserimenti in questi alberi richiedono operazioni di rotazione
- Tutte le operazioni come ricerca, inserimento e cancellazione avvengono in θ(logn)
- Strutture efficienti per i dizionari

ALBERI 2-3

- un albero 2-3 è un albero in cui
    - ogni nodo interno ha 2 o 3 figli
    - tutte le foglie si trovano allo stesso livello
- l'altezza è logaritmica rispetto al numero di foglie: θ(logn)
- i dati vengono memorizzati esclusivamente nelle foglie in ordine crescente da sinistra a destra
- i nodi interni contengono alcune chiavi utilizzate per instradamento e guidare la ricerca:
    - se un nodo ha due figli memorizzerà solamente la chiave piu grande del sottoalbero sinistro
    - se un nodo ha tre figli memorizzerà la chiave piu grande del sottoalbero sinistro e la chiave piu grande del sottoalbero centrale
- Per gli inserimenti bisogna spesso effettuare una operazione di split. Se un inserimento porta un nodo ad avere piu di tre figli bisogna splittare il sotto albero e definire una nuova radice. Ad ogni inserimento tendenzialmente si cresce verso l'alto
- Tutte le operazioni come ricerca, inserimento e cancellazione avvengono in θ(logn)

B – ALBERI

- è definito da un parametro **t**
- un B-albero di ordine t (grado minimo t) è un albero T tale che:
    - ogni nodo interno ha al massimo 2t figli
    - ogni nodo interno ha almeno t figli
    - la radice ha almeno due figli
    - tutte le foglie si trovano allo stesso livello



- Ogni foglia contiene k chiavi ordinate a1<=a1<=…<=ak
- Ogni nodo interno con k+1 figli e sottoalberi T0,…,Tk contiene k chiavi ordinate tali che per ogni chiave ci di Ti, risulta c0<=a1<=c1<=a2<=…<=ck-1<=ak<=ck
- Sembra complicato ma in realtà è una stupidata:
- L'inserimento è una rottura perché bisogna anche qua effettuare operazioni di split nel caso non ci fosse piu spazio. (si dividono in 3 nodi di chiavi, t-1 da una parte, 1 al centro e t dall'altro lato)
- **RICERCA:** tempo θ(logn) e accessi a memoria asintotici a $\log_t n$
- **INS e CANCELLAZIONE:** tempo θ(t logn) e accessi a memoria asintotici a $C*\log_t n$ (C è una costante piccola)

MERGE SORT

- **Struttura dati utilizzata**: array
- **tempo algoritmo**:
  - **caso migliore**: $O(n\log_2 n)$ (la partizione è bilanciata)
  - **caso peggiore**: $\theta(n^2)$ (l'array è già ordinato)
  - **caso medio**: Numero confronti pari a $1.39n\log_2 n$
- **Spazio occupato**:
  - **Caso migliore:** $O(\log_2 n)$ suddivisione bilanciata
  - **Caso peggiore:** $\theta(n)$ suddivisione sbilanciata
- **Descrizione problema e spiegazione della soluzione**:
  Si supponga di avere un array da ordinare, si sceglie un valore chiamato pivot arbitrario. L'array viene diviso cosi: da un lato si mettono tutti gli elementi minori del pivot e dall'altra si mettono tutti quelli maggiori. Successivamente si fa una concatenazione dei due array.
  Nel quicksort è importante la parte del partizionamento . L'algoritmo alla fine divide l'array in 3 parti, la prima composta dai numeri minori uguali al pivot (anche non in ordine), poi il pivot stesso e i numeri maggiori del pivot (anche non in ordine). Infine l'algoritmo restituisce l'indice del pivot.
- **Descrizione algoritmo sommaria:**
  L'algoritmo partiziona è fondamentale che abbia pochi confronti. Si inizia scegliendo come pivot il primo elemento dell'array e poi si utilizzano due indici per definire il partizionamento. Il primo indice parte da sinistra, e il secondo indice parte da destra. Si muove con l'indice di destra e lo si ferma quando si incontra un valore minore del perno. Poi si parte da sinistra e si ferma quando si trova un elemento maggiore del perno. Quando li trovo, bisogna scambiarli. L'algoritmo continua in questo modo scorrendo prima da destra e poi da sinistra fino a quando l'indice di destra non diventa minore dell'indice di sinistra. Come ultima operazione bisogna sostituire il pivot (primo elemento) con l'elemento sotto indice destro.
  La procedura quicksort utilizza la tecnica del merge, invoca la funizona partiziona che restituisce l'indice del pivot nella sua posizione finale, poi successivamente chiama ricorsivamente se stessa due volte utilizzando come estremi di partizionamento i ed m per la prima e poi m+1 ed f per la seconda.
- **Considerazioni**:
  - è possibile effettuare una versione migliore per quanto riguarda lo spazio che porta ad avere nel caso peggiore $\theta(\log n)$ spazio necessario

MERGE SORT

- **Struttura dati utilizzata**: array
- **tempo algoritmo**: θ(nlogn)
- **Spazio occupato**: θ(n)
- **Descrizione problema e spiegazione della soluzione**:
  Merge: Dati due array già ordinati, l'algoritmo di merge necessita di un terzo array di dimensione uguale alla somma delle due dimensioni degli array da unire.
  Ad ogni passo l'algoritmo analizza l'elemento piu piccolo di entrambi gli array e seleziona il piu piccolo tra i due disponibili, lo inserisce poi in posizione k nell'array di merge. Successivamente viene aggiornato l'indice k e l'indice del vettore del quale è stato scelto il valore minore. Quando l'algoritmo arriva alla fine di uno dei due array, copia tutto il contenuto restante dell'altro array all'interno del vettore di merge.
  Una volta definita l'operazione di merge, l'algoritmo di merge Sort prevede la divisione dell'array in due parti ordinate e poi la chiamata alla procedura di merge per unire tutte le soluzioni
- **Descrizione algoritmo sommaria:**
  L'algoritmo di mergeSort prevede l'utilizzo della tecnica Merge. Impiega la ricorsione e la tecnica divide et impera per la divisione e l'ordinamento dei due array.
  Si costruiscono due array, ordino la prima parte e la seconda parte in modo ricorsivo e si effettua infine l'operazione di merge dei due array. Essendo che in queste condizioni vi è un eccessivo utilizzo di stack, è possibile definire una versione migliore che impiega due array e due indici che definiscono inizio (i) e fine (f) della porzione da ordinare. Successivamente calcolo il punto medio della parte da ordinare ((f-i)/2) e richiamo ricorsivamente la procedura due volte, la prima passandole entrambi gli array e come parametri di indici il valore di i ed m e la seconda passandole i due array e come indici m ed f. Dopo aver effettuato le chiamate ricorsive effettuo una chiamata merge passandole entrambi gli array e gli indici i,m ed f
- **Considerazioni**:
  - Merge richiede n-1 confronti
  - Il tempo del mergesort ricorsivo è un macello, ad ogni livello della ricorsione ho dei nuovi array quindi diventa uno spreco disastroso.
  - L'algoritmo va spiegato in modo RICORSIVO e non ITERATIVO!

HEAP SORT

- **Struttura dati utilizzata**: vettore posizionale
- **tempo algoritmo**: θ (nlogn)
- **Spazio occupato**: costante
- **Descrizione problema e spiegazione della soluzione**:
  La descrizione embrionale dello heapsort è la seguente ed impieaga una struttura dati aggiuntiva, ovvero una lista.
  la prima cosa da fare è creare uno heap a partire dall'array, finchè non ho svuotato lo heap devo per ogni iterazione rimuovere da H il valore della radice ed aggiungerlo all'inizio della lista X. Successivamente bisogna rimuovere la foglia piu a destra dell'ultimo livello e collocare il valore nella radice. Infine bisogna effettuare un'operazione di risistemazione dello heap.
- **Descrizione algoritmo sommaria:**
  L'algoritmo Heapsort implementato in loco utilizza solamente un array A.
  La prima cosa da fare è creare uno heap a partire da A. Una volta che l'array A è uno heap, e per ogni indice l a partire da n-1 fino a 1, si scambia A in posizione 0 con A in posizione l e poi si chiama la procedura risistema passando come parametro alla procedura l'array A,la radice zero e l
- **Considerazioni**:
  - L'albero binario quasi completo si definisce tramite vettore posizionale. Dato un nodo in posizione i sappiamo che i suoi figli sono in posizione 2i+1 e 2i+2
  - La procedura risistema riceve come parametri l'array A e due indici r ed l, r indica la radice mentre l indica la prima posizione che non è dello heap
  - La procedura creaHeap parte dal nodo n-1 e va all'indietro fino a zero e ogni volta chiama la procedura risistema passando come parametri A (l'array), i (indice corrente) e n. Quello che fa praticamente è andare dal fondo a leggere l'albero in orizzontale per livello e man mano chiama su ogni nodo risistema cosi da man mano sistemare l'albero in sottoalberi e successivamente risistemare tutto e creare dunque uno heap
  - L'algoritmo è un algoritmo in loco
  - Tramite creaHeap bottom-up non serve memoria aggiuntiva
  - Metodo NON stabile
  - *"- l'heap è un albero binario con la caratteristica importante per cui la chiave dei nodi padre è in una relazione di maggiore o minore con i nodi figli in base al tipo di heap considerato. Nel caso di un MaxHeap il padre avrà sempre un valore del nodo superiore ai figli, nel caso invece di MinHeap avrà un valore inferiore a quello dei figli. Altra particolarità è quella di parlare di un albero binario quasi completo. Questa caratteristica è importante poiché garantisce una altezza logaritmica e quindi una forma compatta per gli algoritmi che manipolano gli heap. Gli heap vengono utilizzati per diverse strategie come ad esempio l'ordinamento HeapSort oppure le code con priorità"*

INTEGER SORT

- Si utilizza un array di dimensione n inizializzato a zero
- Ogni volta che viene inserito un valore x viene incrementato il valore dell'array in posizione x
- È utile solamente per gli interi, è un algoritmo banale

BUCKET SORT

- Essendo integerSort utile solamente per gli interi, il bucketSort risolve questo problema permettendo di associare ad ogni chiave intera una lista concatenata.
- In questo modo tutti quanti i nodi di valore x verranno inseriti nella lista presente in posizione x dell'array di liste.
- All'array di dimensione b vengono associati b secchi e poi si inserisce nel secchio giusto l'elemento che voglio inserire:
    - Si crea un array di bucket, ovvero un array di liste alla fine
    - Si scandisce l'array e si inserisce ogni elemento all'interno del secchio corretto
- Per effettuare effettivamente l'ordinamento mi basta versare i bucket ovvero scorrere l'array e per ogni posizione inserire in un nuovo array di elementi gli elementi di ogni bucket in ordine nel quale son presenti nel bucket.
- Il bucket Sort richiede **tempo** $\theta(b+n)$ dove b è la lunghezza dell'array di bucket, e n è la lunghezza massima della lista di un bucket:
    - Se b è O(1) allora il tempo è $\theta(n)$
    - Se b è $\theta(n^2)$ allora il tempo è $\theta(n^2)$
- È un algoritmo **NON in loco**
- È un algoritmo **STABILE** → mantiene l'ordine interno e questo aiuta nel caso anche di molteplici sort
- è possibile utilizzare questo algoritmo anche solo per ordinare le liste tramite puntatore. Ci vorrebbe sempre l'utilizzo di un array ma nulla in piu.
- È possibile effettuare piu bucketSort su una lista di elementi, ordinando magari prima per il giorno e dopo per il mese etc …

RADIX SORT

- Nel caso di bucketSort se b=5000 non ne esco piu, è necessario utilizzare quindi un algoritmo diverso
- Il radixSort è un insieme di operazioni di bucketSort, nel caso di interi si parte dalle unità e si effettua il bucketsort, poi si passa alle decine e si effettua il bucket sort e cosi via fino alla fine.
- Non per forza serve utilizzare una base decimale ma vanno bene tutte le basi. Bisogna modificare un po' l'algoritmo ma il concetto è sempre lo stesso:
    - Es: se devo ordinare n chiavi tra 0 e 999999999 con base = 10^3 bastano 3 passate di bucketSort, poiché 999999999 se diviso in gruppi di tre: 999 999 999
    - Es: se devo ordinare n chiavi su 32 bit ovvero tra 0 e 2^{32}-1:
        - Con base = 2^8, servono 4 passate di bucketSort
        - Con base = 2^{16}, servono 2 passate di bucketsort
    - Per ottenere la cifra di posizione t di x in base b bisogna fare $\boldsymbol{(x/b^t)\ MOD\ b}$ cosa che in binario si fa molto velocemente
- **Tempo:** $\theta(n)$

FLOYD WARSHALL

- **Struttura dati utilizzata**: matrice dei pesi
- **Serve a**: calcolare i cammini minimi tra ogni coppia di vertice
- **tempo algoritmo**: O(n^3)
- **Spazio occupato**: O(n^3) si può portare a O(n^2)
- **Descrizione problema e spiegazione della soluzione**:

  Floyd warshall è un algoritmo che impiega la tenica della programmazione dinamica.
  Si suppongono un insieme di vertici e dij il peso del cammino minimo da vi a vj. Si parla di un cammino k vincolato dove k rappresenta un valore che va da 0 ad n. Nella distanza da i a j è consentito passare solamente tra vertici con indice minore o uguale a k. Primo e ultimo possono anche essere k ma quelli intermedi devono avere indice minore o uguale a k, quelli maggiori sono vietati. Se k è uguale ad n significa che posso andare ovunque, se k è uguale a zero non posso andare da nessuna parte. (il vincolo è sull'indice dei vertiti intermedi e non sul numero di vertici intermedi). Per risolvere il problema si parte da k uguale a zero e pian piano si aumenta k, cosi facendo si permettono piu passaggi all'aumentare di k.
  Si parte da k uguale a zero come detto, e questo è il caso base, per k maggiore di zero bisogna valutare ogni volta se conviene tenere la strada vecchia o seguire la strada nuova concessa dal fatto che k è aumentato (e quindi permette il passaggio da piu vertici)
  Nell'algoritmo, si utilizzeranno n matrici dove la matrice Dk conterrà i valori del passo k. Inizialmente quindi si inizializza la matrice D0. Si inizializza nel seguente modo, se i è ugulae a j allora D0[i,j] è uguale a zero, se invece c'è un arco allora si mette il valore del peso dell'arco tra i due vertici. Successivamente si riempiono le altre matrici per k=1 a k=n. Per riempirle devo guardare le matrice precedenti, e vedo, se Dk-1[i,k]+D[k-1] è minore di Dk-1[i,j] allora Dk[i,j] viene aggiornata al nuovo valore, altrimenti no.

- **Descrizione algoritmo sommaria:**

  Per ricostruire il cammino è necessario utilizzare una matrice ausiliaria P, inizializzata a zero inizialmente, quando scopro che la strada per andare da i a j passa per k allora io aggiorno la matrice P in posizione i j a k. In questo modo abbiamo una matrice dei massimi indici dei vertici del cammino minimo tra due vertici. Nel caso volessi trovare il cammino minimo tra i vertici V1 e V3 allora dovrei andare a vedere nella matrice P il valore in P[i,j] dove i è uguale a v1 e j uguale a v3; il numero presente nella posizione mi dirà il vertice precedente, e cosi torno indietro. Nel caso trovassi in una posizione P[i,j] il valore zero allora significherebbe che io non sto considerando un vertice ma un arco.

- **Considerazioni**:
  - Non posso andare sotto O(n^2) di spazio sicuramente, poiché l'output è una matrice e non può essere meno di n^2
  - Importante ricordare che con FW se ci sono pesi negativi ma non ci sono cicli ci peso negativo allora tra ogni coppia di vertici esiste un cammino minimo semplice.

BELLMAN FORD

- **Struttura dati utilizzata**: array
  Serve a: trovare i cammini minimi da un vertice a tutti gli altri.
- **tempo algoritmo:** O(n*m) dove n numero vertici e m numero archi
- **Spazio occupato:** costante
- **Descrizione problema e spiegazione della soluzione**:
  Con l'algoritmo di Bellman-Ford si cerca sempre di trovare i cammini minimi da un vertice a tutti gli altri.
  Bellman Ford utilizza un aproccio di programmazione dinamica come FloydWarshall. Si ha una sorgente e si considera un vertive v, si dice che il cammino è k-vincolato dove k a differenza di floyd warshall significa il numero massimo di archi disponibile tra la sorgente e il vertice.
  La restrizione non è piu su quali vertici visitare ma su quanti vertici posso visitare. Nel caso base ho k=0 quindi non posso visitare alcun arco, di conseguenza se ho che V è uguale alla sorgente allora ho distanza uguale a zero, infinito altrimenti.
  Per k maggiore di 0 allora il calcolo della distanza k vincolata è fatta in funzione delle distanze precedenti.Conosciamo il costo di un cammino con k-1 archi, per k quindi so che posso aggiungere un arco e vado a vedere se la nuova strada definita dall'aggiunta del nuovo arco diminuisce o meno. L'idea è che arrivando ad n abbiamo esplorato tutti quanti i cammini piu corti.

- **Descrizione algoritmo sommaria:**
  Si considera d un vettore con indivi in V, si inizializza il vettore secondo la regola definita precedentemente.
  Successivamente provo a costruire cammini sempre piu lunghi: da k uguale ad uno a k guale a n-1 (ovvero nel momento in cui aggiunto un arco). Prendo un arco u,v, supponiamo di avere un arco che arriva da s ad u e prolungo il cammino, e verifico se il nuovo cammino ha un costo inferiore del precedente. Se costa di meno aggiorno d[v]
- **Considerazioni**:

  importante: in n-1 passi si propagano tutti quanti i pesi e si aggiornano

DIJKSTRA

- **Struttura dati utilizzata**:
  Serve a: trovare i cammini minimi da un vertice a tutti gli altri.
- **tempo algoritmo:** O(mlogn) dove m è il numero di archi
- **Spazio occupato:** costante
- **Descrizione problema e spiegazione della soluzione**:
  è piu efficiente di bellman ford ma non lavora con pesi negativi.
  utilizza uno schema greedy, associando ai vertici delle distanze 0 al vertice di partenza e infinito agli altri. Ad ogni iterazione tramite una selezione greedy l'algoritmo si concentra su un vertice.
  Viene scelto il vertice con la distanza minima dalla lista di vertici candidati e da quel vertice scelto "u" va a vedere tutti i vertici collegati. Per ogni vertice v raggiunto da "u" va a vedere se la distanza presente nel vettore delle distanze è superiore a quella ottenibile dal nuovo percorso. In caso affermativo bisogna aggiornare il vettore delle distanze
  Per ricavare i cammini bisogna percorrere a ritroso, ovvero se su un vertice "u" ho un determinato valore di distanza, controllo tutti gli archi entranti e vedo se la distanza di "u" meno il peso dell'arco entrante è uguale al valore dell'arco uscente dal vertice da cui parte di fatto l'arco che entra in "u"

- **Descrizione algoritmo sommaria:**
  si inizializza un vettore delle distanze tramite le regole definite in precedenza mettendo a zero la distanza del nodo S da V se S è uguale a V e inserendo infinito altrimenti.
  Successivamente avviene la scelta Greedy ovvero si sceglie l'elemento con peso (distanza) minore dalla lista di vertici, si vanno a prendere tutti i suoi archi uscenti e si verifica se aggiornare o meno i vertici raggiunti. Una volta fatta questa scelta si rimuove il nodo appena analizzato dalla lista di vertici candidati alla scelta greedy
  Per poter scegliere la distanza minima dobbiamo utilizzare una coda con priorità.
  Usandola quello che bisogna fare è inizializzarla all'inizio con tutti i vertici V e associando a loro come parametro di priorità la distanza. Quando bisogna prelevare l'elemento minimo bisogna fare una operazione di deleteMin() e per aggiornare la coda con priorità nel caso la distanza di un vertice sia effettivamente minore bisogna utilizzare un changeKey()
- **Considerazioni**:
  - \- si fanno n deleteMin che costano O(logn)
  - si fanno al piu m changeKey che costano O(logn)
  - inizializzare la coda con priorità costa O(n) e il vettore degli indici di V sempre O(n)

PRIM

- **Struttura dati utilizzata**: Coda con priorità
- **Serve a**: trovare l'albero ricoprente minimo di un grafo
- **tempo algoritmo**: O(mlogn) uguale a quello di kruskal
- **Spazio occupato**: costante
- **Descrizione problema e spiegazione della soluzione**:
  Si consideri un grafo non orientato connesso. Partendo da un vertice qualunque v si osservano tutti quanti i vertici a lui connessi e si sceglie quello con arco di peso minimo. Si introduce il concetto di frontiera, ovvero tutti gli archi uscenti dalla foresta che si sta costruendo. La frontiera serve per definire successivamente quale arco andare a prendere poiché analizzandoli come per il primo vertice bisogna scegliere sempre quello con peso minimo. Continuando così si arriva a definire un albero ricoprente minimo
- **Descrizione algoritmo sommaria:**
  Inizialmente si definisce T come un albero costituito da un unico vertice v, poi ad ogni vertice v non ancora in T calcoliamo il minimo peso di un arco che va da v a vertici in T e si inserisce in un array delle distanze d in posizione v. Successivamente serve anche un array che definisca il vicino di v collegato dall'arco con peso minore, quindi si utilizza un altro array di nome vicino.
  Si utilizza una coda con priorità che ad ogni vertice assegna come priorità il valore di "d"
  Man mano che la frontiera si espande consulto la "tabella" definita da "d" e "vicino". Una volta che scelgo il minimo dalla tabella, devo aggiornare il valore della priorità dei vertici connessi al vertice in considerazione.
  Guardo gli archi che escono dal vertice y in considerazione, se l'altro estremo è fuori dall'albero vuol dire che l'arco è sul confine e allora mi chiedo se il costo dell'arco è minore del costo dell'arco che avevo prima nella tabella. In caso positivo aggiorno la tabella con il nuovo vicino, la nuova priorità e il vicino mettendo come vicino il nodo y in considerazione. Si procede in questo modo finchè non si svuota tutta la coda di priorità
- **Considerazioni**:
  - Il deleteMin (rimozione dalla coda con priorità) costa O(logn)
  - changeKey costa O(logn)

KRUSKAL

- **Struttura dati utilizzata**: union-find e lista di archi (per rappresentare il grafo)
- **Serve a**: Determinare l'albero ricoprente minimo
- **tempo algoritmo**: O(mlogn)
- **Spazio occupato**: costante
- **Descrizione problema e spiegazione della soluzione**:

  L'algoritmo di Kruskal utilizza la tecnica greedy. Dato un grafo non orientato e pesato, l'algoritmo ordina gli archi dal piu piccolo al piu grande in base ai pesi. Secondo questo ordine, esamino tutti gli archi e verifico, se i due vertici che compongono l'arco non sono connessi nella foresta, allora aggiungo alla foresta l'arco in esaminazione

- **Descrizione algoritmo sommaria:**
  per potere effettuare facilmente le operazioni di controllo di appartenenza di un arco nella foresta dobbiamo utilizzare una struttura dati UNION FIND. Inizialmente si ha una partizione composta solamente dai vertici del grafo e non ci sono connessioni tra questi vertici. Per scoprire se due vertici sono connessi si osserva la partizione e si va a vedere se appartengono allo stesso elemento della partizione. Se non sono connessi non si tocca il grafo, se sono connessi si aggiunge l'arco. Quando si aggiunge l'arco si modifica la partizione: unisco gli insiemi corrispondenti. Si possono quindi utilizzare le operazioni makeSet (costruire la partizione) per fare il controllo si usa FIND e per unire si utilizza UNION
- **Considerazioni**:
  - Il grafo si rappresenta con una lista di archi, la UNION FIND serve come struttura di supporto per organizzare la foresta
  - L'ordinamento si utilizza heapSort e sortando sugli archi, heapsort richiede O(mlogm)

CENNI SU P ed NP

- per **classe di complessità** ci si riferisce all'insieme di problemi che possono essere risolti utilizzando la stessa quantità di una determinata risorsa
- **P** = classe dei problemi che possono essere risolti in tempo polinomiale rispetto alla lunghezza dell'input. Implica diverse tipologie di problemi: ricerca, ottimizzazione, decisione
- **PTIME** = classe dei problemi di decisione risolubili in tempo polinomiale rispetto alla lunghezza dell'input
- **PSPACE** = classe dei problemi di decisione risolubili in spazio polinomiale rispetto alla lunghezza dell'input
- **EXPTIME** = classe dei problemi di decisione risolubili in tempo esponenziale rispetto alla lunghezza dell'input
- RELAZIONI:
    - $P \subseteq EXPTIME$
    - $P \subseteq PSPACE$
    - $PSPACE \subseteq EXPTIME$
    - $P \subseteq PSPACE$ $P \subseteq EXPTIME$
- **NP** = classe dei problemi di decisione risolubili in tempo polinomiale da algoritmi non deterministici
- i problemi NP hanno algoritmi che possono risolverli secondo due fasi:
    - **Fase non deterministica** in cui avviene la costruzione del certificato
    - **Fase deterministica** in cui avviene la verifica del certificato in tempo polinomiale
- Alcuni problemi della classe **NP:**
    1) **PARTIZIONE**
        - Si consideri un insieme finito A di oggetti e una funzione di peso che associa ad ogni oggetto un peso.
        - Esiste un sottoinsieme degli oggetti tale che la somma dei valori degli oggetti contenuti nel sottoinsieme è uguale alla somma dei valori che sta al di fuori del sottoinsieme?
    2) **SODD**
        - Eiste un assegnamento ad un insieme di variabili V tale che la formula booleana sia true?
        - bisogna provare tutte le possibili permutazioni di valori alle variabili per capirlo
        - si cerca di indovinare con la funzione indovina in un range di valori
    3) **CLIQUE**
        - Esiste un sottografo completo di G con k vertici? dove k è un valore intero maggiore uguale a zero

- un problema P è detto **NP-HARD** se ogni problema appartenente alla classe NP è riducibile polinomialmente a P
- un problema P è detto **NP-COMPLETO** se P è NP hard e anche appartenente ad NP. Alcuni sono:
    - SODD (Teorema di **cook**)
    - Cammino semplice piu lungo in un grafo di lunghezza maggiore uguale a k
    - Cammino hamiltoniano: Esiste cammino che visita tutti i vertici del grafo una e una sola volta?
    - Partizione
    - Clique

- **tabella di hash**: collezione di elementi ciascuno dei quali identificato da una chiave, memorizzati in un array.
- la posizione viene definita da una funzione detta di hash
- **funzione di hash** = funzione che trasforma le chiavi in indici
    - una funzione di hash **perfetta** è una funzione di hash iniettiva.
- il **fattore di carico** è un indice definito da **n/m** dove
    - n numero di elementi memorizzati nella tabella
    - posizioni disponibili nella tabella
- nelle tabelle di hash avvengono delle collisioni quindi bisogna:
    - fare in modo che avvengano raramente sparpagliando bene con h le chiavi
    - avere una strategia per gestirle
- una buona funzione hash deve essere
    - veloce da calcolare
    - uniformità ovvero che ogni cella della tabella ha la stessa probabilità di contenere un dato elemento

- GESTIONE DELLE COLLISIONI **ESTERNE**:
    - liste di collisione = se serve inserire dei valori aventi la stessa chiave (definita dalla funzione di hash)
    - la lunghezza media della lista sarà uguale a n/m dove n sono numero di elementi della tabella, ed m il numero di posizioni (uguale al **fattore di carico**).
    - se le chiavi si sparpagliano male allora degenera e diventa come utilizzare una lista concatenata
    - **Agglomerazione** = presenza di zone in cui si concentrano molti dati. Se la funzione hash non sparpaglia bene allora si potrebbero formare alcune liste lunghe
- GESTIONE DELLE COLLISIONI **INTERNE:**
    - indirizzamento aperto
        - se h(k) è una posizione già occupata allora si cerca un'altra posizione libera utilizzando una strategia predefinita.
        - spesso con indirizzamento aperto si utilizza una funzione ausiliaria che è definita dalla chiave e dall'ordine nella scansione. Questa mi da la strategia per capire come scandire la lista:
            - Scansione lineare: si cerca la prima posizione libera e si inserisce l'elemento essa puo portare ad una **agglomerazione primaria** (nuvoloni di dati concentrati in una zona)
                - → **c(k,i) = (h(k) + i) MOD (dim_tabella)**
            - Scansione quadratica: tramite una funzione riduce l'agglomerazione ma puo comunque portare ad una **agglomerazione secondaria**
                - →**c(k,i) = (h(k) + c1i+c2i) MOD (dim_tabella)**
            - hashing doppio : impiega una seconda funzione di hashing per determinare dove inserire i valori aventi stessa chiave. Importante considerare che le due funzioni dovrebbero essere abbastanza diverse in modo da garantire h1!=h2. come per la scansione quadratica potrebbe formarsi una agglomerazione secondaria.
                - → **c(k,i) = (h(k) + i * g(k)) MOD (dim_tabella)**
        - Per cercare un elemento si scansiona fino a quando la tabella non raggiunge un elemento vuoto.
        - per cancellare bisogna utilizzare una **cancellazione logica**, ovvero inserire un bit al record che dice che quella posizione è stata cancellata. Non la cancello effettivamente perché senno la ricerca spiegata prima non funzionerebbe piu
- Quando la tabella viene riempita oltre la soglia stabilita viene creata una tabella piu grande (generalmente il doppio) in cui vengono trasferiti tutti gli elementi. CI sono dei problemi pero':
    - Funzione di hash per la nuova tabella:
        - per sistemare bisogna scegliere una funzione di hash in grado di gestire eventualmente dei rehashing. Generlmanete la funzione di hash è parametrica rispetto alla dimensione della tabella.
    - Costo in termini di tempo (traslocare costa)
- il rehashing ha un costo non da poco quindi, ma è una forma di investimento ovvero quando si effettua un rehashing, poi non dovrò farlo per molto tempo e quindi si può fare una analisi ammortizzata