MASTER THEOREM

$$T(n) = \begin{cases} 1 & n \leq 1 \\ aT(n/b) + cn^{\beta} & n > 1 \end{cases}$$

- Se $\frac{\log a}{\log b} = \alpha > \beta$ $\quad T(n) = \Theta(n^{\alpha})$
- $\alpha = \beta$ $\quad T(n) = \Theta(n^{\alpha} \log n)$
- $\alpha < \beta$ $\quad T(n) = \Theta(n^{\beta})$

ORDINAMENTO

- Selection sort

Partendo da SX, seleziona il valore minore dell' array verso DX e lo scambia con il valore all' indice attuale

```
function SelSort(A[1..n])
    for i=1 to n
        min = i
        for j=i to n
            if A[j] < A[min] then
                min = j
        SWAP(A, i, min)
```

Ogni ciclo di j viene raccorciato di un elemento

$$T(n) = n + (n-1) + \dots + 1 = \sum_{K=1}^{n} K = \frac{n(n+1)}{2} = \Theta(n^2)$$

Quindi nel caso migliore/medio/pessimo il selection sort è sempre $\Theta(n^2)$. È in place ma non stabile.

- Insertion Sort

Partendo da sx, selezioniamo il valore e lo inseriamo spostandolo verso sx finché è maggiore o uguale al valore precedente

```
function Insertion Sort (A[1..n])
    for i = 1 to n
        j = i
        while j > 0 and A[j] < A[j-1]
            SWAP(A, j, j-1)
            j = j - 1
```

Nel caso ottimo (array già ordinato) il ciclo while non viene eseguito, quindi $T(n) = \Theta(n)$. Si può dimostrare che questo accade anche quando l'array è ordinato tranne K elementi, con $K = \Theta(1)$.

Nel caso pessimo (array ordinato al contrario) il costo è uguale al Selection Sort $= \Theta(n^2)$.

Nel caso medio assumiamo che il ciclo while faccia $\frac{(i-1)}{2}$ cicli, quindi

$$T(n) = \sum_{i=2}^{n} \frac{i-1}{2} = \frac{1}{2} \sum_{i=1}^{n-1} i = \frac{1}{2} \frac{n(n-1)}{2} = \Theta(n^2)$$

- **Merge Sort**

Algoritmo divide-et-impera:

- Divide: divide l'array in due metà
- Conquer: chiama merge sort ricorsivamente per ordinare le due metà
- Combine: ricombina le due metà ordinate inserendo da Sx a Dx l'elemento minore fra i due array

```
function MergeSort (A[1..n], l, r)
    if l ≥ r then
        return
    else
        q = ⌊(l+r)/2⌋
        MergeSort (A, l, q)
        MergeSort (A, q+1, r)
        merge (A, l, q, r)

function merge (A[1..n], l, q, r)

    LET B[1..r-l+1] BE A NEW ARRAY
    i = l
    j = q+1
    K = 1
    while i ≤ q and j ≤ r          ] Finché si svuota uno
        if A[i] ≤ A[j] then        ] degli array, si confronta
            B[k] = A[i]            ] il valore degli elem.
```

```
            i++
        else
            B[K] = A[j]
            j++
        K++
    while i ≤ q
        B[K] = A[i]
        i++
        K++
    while j ≤ r
        B[K] = A[j]
        j++
        K++
    for K = 1 to r-l+1
        A[l+K-1] = B[K]
```

minimi dei due array e il minore si inserisce in B

Svuota il resto dell'array non vuoto (non sappiamo qual'è quindi proviamo entrambi)

ricopia $B[1..r-l+1]$ in $A[l..r]$

Analizziamo prima il merge. I cicli while combinati fanno $r-l+1 = n$ cicli e l'ultimo for fa n cicli, quindi $T(n) = \Theta(n)$.

Il merge sort è definito dalla seguente eq. di ricorrenza:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n/2) + f(n) & n > 1 \end{cases}$$

Dove $f(n) = \Theta(n)$ è il costo di merge. Quindi $T(n) = (n \log n)$

- <u>Quick Sort</u>

Algoritmo divide-et-impera:

- Divide:

  scelto un valore qualunque dell'array, si separano tutti i valori minori o uguali e quelli maggiori

- Conquer:

  i due array vengono ordinati ricorsivamente

- Combine:

  l'array ordinato minore viene messo prima del perno e l'array maggiore dopo

```
function QuickSort(A[1..n], l, r)
    if r ≤ l then
        return
    else
        q = DIVIDE(A, l, r)
        QuickSort(A, l, q)
        QuickSort(A, q+1, r)

function DIVIDE(A[1..n], l, r) -> Intero
    p = A[r]        --> scegliamo come perno l'ultimo elemento
    i = l - 1
    for j = l to r - 1
        if A[j] ≤ p
            i = i + 1
            SWAP(A, i, j)
    SWAP(A, i + 1, r)
    return i + 1
```

Caso pessimo: divisione sbilanciata (uno dei due array vuoto, e' altro $n-1$)

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 1+T(n-1)+f(n) & n>1 \end{cases}$$

Dove $f(n)$ è il costo di DIVIDE $= \Theta(n)$

$$T(n) = T(n-1)+1+n =$$
$$= T(n-2)+2+(n-1)+n =$$
$$\vdots$$
$$= T(n-i)+i+\sum_{k=0}^{i-1} n-k$$

La ricorsione termina per $n-i=1$ quindi per $i=n-1$, quindi la ricorsione nel caso pessimo è:

$$T(n) = n + \sum_{k=0}^{n-2} n-k = \Theta(n^2)$$

Nel caso ottimo come mergesort $\Theta(n\log n)$ (anche nel caso medio)

- **Counting Sort**

Se sappiamo il valore massimo possibile di una chiave, contiamo quante volte appaiono tutti i valori nell' array e li reinseriamo nell' array dal valore min al max con la giusta freq.

```
function CountingSort(A[1..n])
    a = MIN(A), b = MAX(A), K = b - a + 1
    LET B[1..K] BE A NEW ARRAY
```

```
for i = 1 to k
    B[i] = 0
for i = 1 to n
    B[A[i] - a + 1]++
j = 1
for i = 1 to k
    for B[i] times do
        A[j] = i + a - 1
        j++
```

Sia $m = b - a + 1$ la cardinalità dell'insieme minore che contiene tutte le chiavi di A.
Il costo comp. é $\Theta(n+m)$, quindi se:

- $m = O(n)$, allora é $\Theta(n)$
- $m = \Theta(n \log n)$, allora é uguale ai divide-et-impera $\Theta(n \log n)$

- <u>Radix Sort</u>

Date chiavi composte da cifre o caratteri, ordina queste prima per la cifra meno significativa poi penultima e così via (con un ordinamento stabile)

<u>Dizionari</u>

	Search	Insert	Delete
BST	$\Theta(\bar{h})$ $\Theta(\bar{h})$	$\Theta(\bar{h})$ $\Theta(h)$	$\Theta(\bar{h})$ $\Theta(h)$
AVL	$\Theta(\log n)$ $\Theta(\log n)$	$\Theta(\log n)$ $\Theta(\log n)$	$\Theta(\log n)$ $\Theta(\log n)$

Hash $O(n)$ $O(1)$ — — -

CODE CON PRIORITÁ

- Binary Min/Max Heap (n = size della coda)
  - insert = $\Theta(\log n)$
  - delete = $\Theta(\log n)$
  - find Min = $\Theta(1)$
  - increase/decrease Key = $\Theta(\log n)$

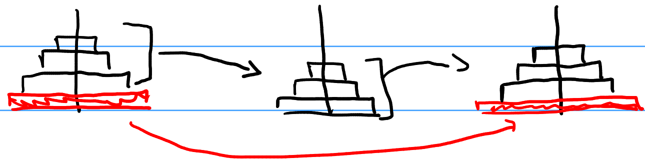
UNION FIND

- makeSet = $O(1)$
- union = QUR $O(1)$; QFP $\Theta(\log n)$
- find = QUR $\Theta(\log n)$; QFP $O(1)$

TECNICHE ALGO

- Divide-et-impera

Binary Search, Merge Sort, Quick Sort

- Torri di Hanoi 

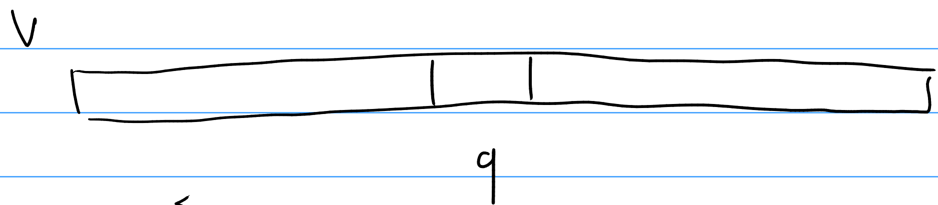
Consideriamo l'ultimo "cerchio" e tutti quelli sopra. Per risolvere il probl. dobbiamo:
- spostare i cerchi sopra su un piolo
- spostare l'ultimo su un'altro piolo
- spostare gli altri cerchi sopra lo stesso piolo

Notiamo che il passo 2 é banale mentre i passi 1 e 3 possono essere risolti chiamando ricorsivamente l'algoritmo

- Moltiplicazione

Il classico algoritmo é $\Theta(n^2)$. Possiamo dividere i due numeri in due parti da $n/2$ cifre da cui possiamo ricavare una formula che contiene 3 prodotti fra numeri con $n/2$ cifre, che ha costo $\Theta(n^{\log 3})$ per risolvere ricorsivamente.

- Sottovettore non-vuoto di valore massimo



- 3 possibilitá:
  - il sottovettore max é tutto nell'area di sx
  - " " di dx
  - il sottovettore contiene $V[q]$

Quindi ValMax = MAX (ValMax(SX), ValMax(DX) o

$S_a + S_b + 1$)

Valore massimo di tutti i sottoarray (anche vuoti) partendo da q

- GRIDDY

  - Resto (quando è sempre possibile!)

    Scelgo sempre la moneta con più valore possibile.

  - Schedoling

    Scelgo sempre il lavoro che ci mette di meno a finire per ottimizzare il tempo medio di attesa

  - Compressione di Huffman

    Inserisco prima i caratteri meno frequenti nell'albero binario (che quindi hanno la codifiche più lunghe) in modo da ottimizzare la dim. del file.

- ## Prog. Dinamica

- Sottovett. massimo

Definiamo $P[i]$ il valore del sottovett. massimo che finisce con $A[i]$. Quindi conoscendo $P[i]$, so che $P[i+1]$ è il massimo fra $P[i]$ e $A[i+1]$. Il sottoproblema banale è $P[1]$ che è $A[1]$. La soluzione del problema è il massimo fra tutti i valori di P.

- Problema dello zaino

Abbiamo un array di oggetti A (che hanno un peso e un valore) e un peso massimo K.
Possiamo creare dei sottoproblemi variando il peso massimo e gli oggetti disponibili →
$P[i,j]$, dove i è l'indice massimo degli oggetti che possiamo usare e j è il peso totale.
$P[i,j] = \max(P[i-1,j], P[i-1, j-A[i]])$, quindi conoscendo $P[a,b]$ $\forall a<i, b\leq j$ posso trovare $P[i,j]$. I sottoproblemi banali sono quando $j=0$ o $i=0$, dove $P[0,j]=P[i,0]=0$

- Seam Carving

Abbiamo una matrice e vogliamo trovare il percorso dalla prima all'ultima riga con costo minore. Consideriamo il sottoproblema $P[0,j]$ come

il costo minimo di un percorso che finisce in $A[i,j]$. Per definizione di percorso, $P[i,j] = \min(P[i-1,j-1], P[i-1,j], P[i-1,j+1]) + A[i,j]$.
se esistono

I sottoproblemi banali sono quando $i=1$, in quel caso $P[1,j] = A[1,j]$. La soluzione al problema è il valore minimo di $P[n,j]$.

- Distanza di Levinstein

Dati due array qual è il numero di "edit" minore per trasformare il primo nel secondo? Definisco il sottoproblema $P[i,j]$ che trasforma i primi i caratteri del primo nei primi j caratteri del secondo. $P[i,j] = \min(P[i-1,j]+1, P[i,j-1]+1, P[i-1,j-1] (+1))$

solo se $A[i] \neq B[j]$

I sottoproblemi banali sono:

- $i=0$ $P[0,j] = j$
- $j=0$ $P[i,0] = i$

I GRAFI

	MATRIX	ADJ. LIST
grado	$O(n)$	$O(\delta(v))$
areAdjacent	$O(1)$	$O(\min(\delta(x), \delta(y)))$
addVertex	$O(n^2)$	$O(1)$
addEdge	$O(1)$	$O(1)$
removeVertex	$O(n^2)$	$O(m)$
removeEdge	$O(1)$	$O(\delta(x)+\delta(y))$

VISITE

- $\boxed{\text{BFS}}$

Visita in ampiezza, si visitano i nodi in ordine di quando vengono scoperti. Più ci fa più ci allontaniamo dalla sorgente

```
function BFS(Grafo G, Vertice s) -> Albero
  for each v ∈ G do  v.visited = false; v.parent = nil
  T := s
  LET Q BE A NEW QUEUE
  Q.enqueue(s)
  s.dist = 0; s.visited = true
  while Q.size ≠ 0
      v = Q.dequeue()                              n
n² o m    for each u adjacent to v do
          if not u.visited
              u.visited = true; u.dist = v.dist + 1
m             Q.enqueue(u)
              T := T ∪ u
              u.parent = v
```

```
        endif
    end for
end while
return T
```

Dato che ogni nodo entra nella coda una sola volta, il ciclo while viene eseguito n volte. Quindi il ciclo for viene eseguito su ogni nodo, e la complessità totale é la somma dei gradi uscenti di tutti i nodi, che é al massimo $2m = \Theta(m)$ PIÚ n-volte la complessita di trovare tutti i nodi incidenti da un altro nodo, che in una matrice é $\Theta(n^2)$ mentre in una ADJ. LIST é $\Theta(n \underbrace{\Sigma(v)}_{m})$, quindi abbiamo $\Theta(n^2)$ o $\Theta(n+m)$.

Ci puó dare il percorso piú breve fra 2 nodi.

$\boxed{\text{DFS}}$

Visita in profonditá, Visita tutti i nodi del grafo (anche se non é connesso) e restituisce un insieme di alberi DFS. Tiene traccia del tempo di "apertura" e di "chiusura" di ogni nodo

```
global time := 0
function DFS (Grafo G) -> Foresta
    LET F BE A SET OF TREES
    for each v ∈ G do (v.status = unexplored;
                       v.parent = nil)
    for each v ∈ G do
        if v.status == unexplored then
            F.add (DFS-visit (G, v))
return F
```

```
function DFS-visit (Grafo G, Vertice v) -> Albero
    LET T BE A NEW TREE; T := v
    v.dt = ++time; v.status = open
    for each u adjacent to v do
        if u.status = unexplored then
            T := T ∪ DFS-visit(G, u)
            u.parent = v
    v.ct = ++time; v.status = closed
    return T
```

Complessità "uguale" a BFS. Lo possiamo usare per identificare DAG e ordinarli i nodi in modo topologico e per individuare componenti connesse (e fortemente connesse)

MST

Ci da l'albero contenente tutti i vertici di un grafo con la minore somma di tutti i pesi degli archi.

- $\boxed{\text{Kruskal}}$

Algoritmo greedy, usa un algo sorting e una struttura UF per evitare di connettere due componenti già connesse, che creerebbe un ciclo

```
function Kruskel(Grafo G) -> Albero
    LET T BE A NEW TREE
    LET UF BE A NEW UNION-FIND
    for i = 1 to V.size                      n
        UF.makeSet(i)
    SORT(E) BY WEIGHT (NON DECREASING)        m log m
    for each e ∈ E do
        us := UF.find(e.source)  ]
        vs := UF.find(e.dest)    ]  m
        if vs ≠ us then
            UF.union(us, vs)     (n-1) log n
            T := T ∪ {u, v}  // aggiungi arco all'albero
    return T
```

$$O(n + m\log n + m + (n-1)\log n) = O(m \log n)$$

- $\boxed{\text{Prim}}$

Scelto un nodo radice viene usata una coda con priorità per avere il prossimo arco a peso più basso che collega un nodo nell'MST con un altro nodo che non appartiene

CAMMINI MINIMI

- **SSP**

Restituiscono tutti i cammini minimi partendo da una sorgente e arrivando a tutti i nodi ragg.

- $\boxed{\text{Bellman - Ford}}$ DP ?

Usa della prop. delle distanze minime per "rilassare" ad ogni iterazione una delle distanze per eccesso inizializzate a $+\infty$. Il ciclo viene ripetuto $n-1$ volte, finché tutte le distanze sono corrette.

Condizione di Bellman:

$\forall$ arco (u, v) e $\forall$ vertice s:

$$d_{sv} \leq d_{su} + w(u, v)$$

Quindi possiamo iterare su ogni arco (u, v) e dire che se $d_{su} + w(u, v) < D_{sv}$, allora $D_{sv} = d_{su} + w(u, v)$ è un'approssimazione migliore. Siamo sicuri che se lo eseguiamo su ogni arco, uno dei rilassamenti sarà il primo passo di rilassamento "corretto". Ripetiamo questo $n-1$ volte (per ogni vertice di dest. possibile da un cammino di s) e sappiamo tutti i valori D_{sv_k} corretti

```
                                           int s           double
function BellFord (Grafo G = (V,E,w)) -> [1..n]
    int n = G.numNodi()
    double D[1..n]
    int pred[1..n]
    for i=1 to n
        D[i] = +∞
        pred[i] = -1
    for i=1 to n-1
        for each (u,v) ∈ E do
            if D[v] > D[u]+w(u,v) then
                D[v] = D[u]+w(u,v)
                pred[v] = u
    // controllo neg.
    for each (u,v) ∈ E do
        if D[v] > D[u]+w(u,v) then
            print "cicli neg!"; return nil
    return D
```

$\Theta(nm)$ → lento, però funziona anche per archi negativi!

- $\boxed{\text{Dijkstra}}$

Algoritmo greedy, funziona solo con archi positivi. Partendo con $T := s$ (radice), $\forall$ arco (u,v) dove $u \in T$ e $v \notin T$ si sceglie l'arco (u,v) che minimizza $d_{su} + w(u,v)$.

```
double[1..n] function Dick (Grafo G=(V,E,w), int s)
 int n := G.numNodi()
 double D[1..n]
 int pred[1..n]
 for i=1 to n
     D[i]=+∞; pred[i]=-1                         O(n)
 D[s]=0
 LET Q BE A NEW MIN PRIORITY QUEUE
 Q.enqueue(s, D[s])
 while not Q.isEmpty()
     u := Q.dequeue()                          O(n log n)
     for each v adj to u do
         if D[v] = +∞ then
             D[v] = D[u] + w(u,v)
             Q.enqueue(v, D[v])
             pred[v]=u                                O(m log n)
         else if D[v] > D[u] + w(u,v) then
             D[v] = D[u] + w(u,v)
             Q.decreaseKey(v, D[v])
             pred[v]=u
 return D
```

O(m log n) ma non funziona con archi negativi

- All pairs SP

· $\boxed{\text{Floyd – Warshall}}$