# Calcolo della complessità

## Metodo generale

> 1. Si "indovina" una possibile soluzione e si formula l'ipotesi

> 2. Si sostituisce le espressioni di $T(\cdot)$ con l'ipotesi induttiva

> 3. Si dimostra la validità nel caso base

## Cheat sheet

> ### Cheat sheet
>
> | forma | Complessità superiore | formula matematica |
> | --- | --- | --- |
> | $T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 1$ | $O(n)$ | $\sum_{i=0}^{log\ n} 2^i$ |
> | $T(\lfloor \frac{n}{2} \rfloor) + n$ | $O(n)$ | $n \cdot \sum_{i=0}^{log\ n} \frac{1}{2}^i$ |
> | $T(n-1) + n$ | $O(n^2)$ | $\sum_{i=1}^{n} i$ |
> | $2T(\lfloor \frac{n}{2} \rfloor) + n$ | $O(n\ log\ n)$ | |
> | $9T(\lfloor \frac{n}{3} \rfloor) + n$ | $O(n^2)$ | |

# Master theorem

## Ricorrenze lineari con partizione bilanciata

Presa una funzione nella forma

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

Posto $\alpha = log_b a$, allora

$$T(n) = \begin{cases} \Theta(n^\alpha) & \alpha > \beta \\ \Theta(n^\alpha log\, n) & \alpha = \beta \\ \Theta(n^\beta) & \alpha < \beta \end{cases}$$

## Ricorrenze lineari di ordine costante

Con $a_{1..h}$ tutte costanti positive:

$$T(n) = \begin{cases} \sum_{i=1}^{h} a_i T(n-i) + n^\beta & n > h \\ \Theta(1) & n \leq h \end{cases}$$

Posto $a = \sum a_{1..i}$ allora

$$T(n) = \begin{cases} \Theta(n^{\beta+1}) & a = 1 \\ \Theta(a^n n^\beta) & a \geq 2 \end{cases}$$

# Algoritmi utili

## Sorting

### Selection

> ```
> void selection_sort(vector<int> &arr){
>         for (int i = 0; i < arr.size(); i++){
>                 int curr_min = min(arr, i);
>                 if(curr_min != -1){
>                         swap(arr,i,curr_min);
>                 }
>         }
> }
> ```

### Insertion

> ```
> void insertion_sort(vector<int> &arr){
>         for(int i = 1; i < arr.size(); i++){
>                 int in_hand = arr[i];
>                 int ii = i;
>                 for(; ii > 0 && arr[ii-1] > in_hand; ii--){
>                         arr[ii] = arr[ii-1];
>                 }
>                 arr[ii] = in_hand;
>         }
> }
> ```

### Merge

```
void merge(vector<int> &arr, int begin, int mid, int end) {
	vector<int> temp;

	int left_index = begin, right_index = mid + 1;

	while (left_index <= mid && right_index <= end) {
		if (arr[left_index] <= arr[right_index]) {
			temp.push_back(arr[left_index]);
			left_index++;
		} else {
			temp.push_back(arr[right_index]);
			right_index++;
		}
	}

	if (left_index <= mid) {
		while (left_index <= mid) {
			temp.push_back(arr[left_index]);
			left_index++;
		}
	}

	for (int el : temp) {
		arr[begin] = el;
		begin++;
	}
}

void merge_sort(vector<int> &arr, int begin, int end) {
	if (begin < end) {
		int mid = (begin + end) / 2;

		merge_sort(arr, begin, mid);
		merge_sort(arr, mid + 1, end);
		merge(arr, begin, mid, end);
	}
}
```

## Quicksort

```
quickSort(Item a[], int low, int high){
        if[low < high]{
                int j = partition(a,low,high)
                quickSort(a,low, j-1)
                quickSort(a, j+1, high)
        }
}

int partition(Item a[], int low, int high){
        Item pivot = a[lo]
        int j = low
        for [i=low+1 to high]{
                if [ a[i]<pivot ] {
                        j++
                        swap(a[i],a[j])
                }
        }
        a[low] = a[j]
        a[j]=pivot
        return j
}
```

## Alberi

### DFS

```
void dfs(){
        // pre-order print()
        if(left != NULL){
                (left)->dfs();
        }
        //in-order
        cout << value << " ";
        if(right != NULL){
                (right)->dfs();
        }
        //post-order print()
}
```

### BFS

```
void bfs(){
        vector<Node*> queue = vector<Node*>();
        queue.push_back(this);

        while (queue.size()>0){
                if((*queue[0]).left != NULL){
                        queue.push_back((*queue[0]).left);
                }
                if((*queue[0]).right != NULL){
                        queue.push_back((*queue[0]).right);
                }
                cout << (*queue[0]).value << " ";
                queue.erase(queue.begin());
        }
        cout << endl;
}
```

**Calcolo delle possibili combinazioni**

> $$P(n) = \begin{cases} 1 & n = 0 \\ \sum_{s=0}^{n-1} P(s) \cdot P(n-s-1) & n \geq 1 \end{cases}$$

> $$T(n) = \begin{cases} 0 & n \; pari \\ 1 & n = 1 \\ \sum_{i=1}^{n-2} T(i) \cdot T(n-1-i) & n \geq 1, \; n \; dispari \end{cases}$$

> $$T(n,k) = \begin{cases} 1 & n = 1, \; k = 0 \\ 0 & k \geq n \\ 2T(n-1,k-1) + \sum_{i=1}^{n-2} T(i,k) \cdot T(n-1-i,k) & else \end{cases}$$

# Grafi

**BFS**:

```
Queue q
q.enqueue(r)
foreach [nodes-r]{
        distance[n] = -1
}
distance[r] = 0

while [q not empty]{
        curr = q.dequeue
        foreach [node adjacent to curr]{
                if distance[adj] == -1{
                        distance[adj] = distance[curr]+1
                        ***
                        q.enqueue(adj)
                }
        }
}
```

**DFS**:

```
visited[curr] = true //visita pre-order
foreach [node adjacent to curr]{
        if [not visited]{
                dfs(Graph, adj, visited) //visita in-order
        }
}
// visita post-order
```

### Identificazione dei cicli

```
bool hasCycleRec(Graph G, Node n, int &time, int[] dt, int[]
ft){
        time++
        dt[n]=time
        foreach[adj in G.adj(n)]{
                if [dt[adj] == 0]{
                        if [hasCycleRec(G,adj,time,dt,ft)]{
                                return true
                        }
                } else if [dt[n]>dt[adj] and ft[v]==0]{
                        return true
                }
        }
        time++
        ft[u]=time
        return false
}
```

> ```
> bool hasCycle(Graph G){
>         bool visited [G.size]
>         foreach [n in G.v]{
>                 visited[n] = false
>         }
>         foreach [n in G.v]{
>                 if [not visited[n]]{
>                         if [hasCylceRec(G,n,NULL, visited)]{
>                                 return true
>                         }
>                 }
>         }
>         return false
> }
> ```

### Per grafo orientato

> ```
> bool hasCycleRec(Graph G, Node n, int &time, int[] dt, int[]
> ft){
>         time++
>         dt[n]=time
>         foreach[adj in G.adj(n)]{
>                 if [dt[adj] == 0]{
>                         if [hasCycleRec(G,adj,time,dt,ft)]{
>                                 return true
>                         }
>                 } else if [dt[n]>dt[adj] and ft[v]==0]{
>                         return true
>                 }
>         }
>         time++
>         ft[u]=time
>         return false
> }
> ```

### Albero di copertura

```
dfs-schema(Graph G, Node n, int &time, int [] dt, int [] ft){
        time++
        dt[n] = time
        foreach [adj in G.adj(n)]{
                if [dt[adj]==0]{
                        dfs-schema(G, adj, time, dt, ft)
                } else if [dt[n]>dt[adj] and ft[adj] == 0]{
                        // arco indietro
                } else if [dt[n]<dt[adj] and ft[adj] != 0]{
                        //arco in avanti
                } else {
                        arco di traversamento
                }
        }
        time++
        ft[n] = time
}
```

### Ordinamento topologico

```
Stack topSort(Graph G){
        Stack s = Stack()
        bool visited [G.size]
        foreach [node in G] {
                visited[node] = false
        }
        foreach [n in G]{
                if [not visited[n]]{
                        ts-dfs(G,n,visited,s)
                }
        }
        return s
}
```

```
ts-dfs(Graph G, Node n, bool[] visited, Stack s){
        visited[n] = true
        foreach[adj in G.adj(n)]{
                if [not visited[adj]]{
                        ts-dfs(G, adj, visited, s)
                }
        }
        s.push(n)
}
```

## Connessioni forti

Per individuare le componenti fortemente connesse possiamo usa l'algoritmo di Kosaraju:

1. applico l'ordinamento topologico sul grafo
2. Generiamo il grafo trasposto $G_t$ (grafo con la direzione degli archi invertita)
3. individuo le componenti connesse usando l'ordinamento topologico al contrario (prendo come argomento lo stack e `pop()` gli elementi finché presenti)
   Le componenti connesse individuate saranno fortemente connesse

# Algoritmi ricorrenti

Navigazione degli alberi

```
Object function(Node n, [dati da passare ai figli]){
        [Processa prima di mandare ai figli]
        obj = function(n.child, data)
        [Processa i dati di ritorno]
        return [dati da tornare ai padri]
}
```

Generazione di alberi

```
tree function(Arr [], int index, int n){
        Tree element = Arr[...]
        tree.insert(function(A, [nuovi indici]))
        return element
}
```

Combinatoria ricorsiva

```
int function(int remainingEl, int possibleEl){
        if(remainingEl == 0){
                return 1
        }
        int count = 0
        for(i from 0 to max(possibleEl, remainingEl)){
                count += function(remainingEl -1, possibleEl)
        }
        return count
}
```

## Combinatoria programmazione dinamica

```
int function(int totEl, int possEl){
        int t[n+1]
        t[0] = 1
        for(i from 1 to n){
                t[i] = 0
                for(k from 1 to minArg(i, possEl)){
                        t[i] += t[i-k]
                }
        }
        return t[n]
}
```

## Ricerca dicotomica

```
int function(Arr A [], int index, int n, int target){
        if(n >= index){
                int m = index + (n - index)/2

                [check elements, eventually return]

                if([condition left]){
                        return function(A, index, m)
                } else {
                        return function(A, m+1, n)
                }
        }
}
```

Indici paralleli

```
int function(Arr A [], int n){
        int max = 0, curr = 0
        int left = 0, right = 0
        while(right < n){
                if([condition]){ // ex: (a[r] - a[l] <= limit)
                        curr++
                        max = maxArg(max, curr)
                        right ++
                } else {
                        curr--
                        left++
                }
        }
        return max
}
```

# Strutture Dati

## Sequenza

> **Funzioni**
>
> - `boolean isEmpty()`
> - `boolean finished(Pos p)` (ritorna true se la posizione è la testa o la coda)
> - `Pos head()`
> - `Pos tail()`
> - `Pos next(Pos p)`
> - `Pos prev(Pos p)`
> - `Pos insert(Pos p, Item v)`
> - `Pos remove(Pos)`
> - `Item read(Pos p)`
> - `write(Pos p, Item v)`

## Insieme

> **Funzioni**
>
> - `int size()`
> - `boolean contains(Item x)`
> - `insert(Item x)`
> - `remove(Item x)`
> - `Set union(Set A, Set B)`
> - `Set intersection(Set A, Set B)`
> - `Set difference(Set A, Set B)`

# Dizionario

> **Funzioni**
>
> Dove `k` è la chiave e `v` l'elemento
> `Item lookup(Item k)`
> `insert(Item k, Item v)`
> `remove(Item k)`

# Albero binario

> **Funzioni del nodo**
>
> `Tree(Item v)` (Costruisce un nuovo nodo con `v` come valore)
> `Item read()` (legge il valore)
> `write(Item v)`
> `Tree parent()` (Ritorna il nodo padre, se esistente)
> `Tree left()`
> `Tree right()`
> `insertLeft(Tree t)`
> `insertRight(Tree t)`
> `deleteLeft()`
> `deleteRight()`

# Albero binario di ricerca

> **Specifica**
>
> Oltre alle normali funzioni di un albero binario abbiamo anche:
>
> - `Item lookup(Item k)`
> - `Tree successorNode(Tree t)`
> - `Tree predecessorNode(Tree t)`
> - `Tree min()`
> - `Tree max()`

### Successore e precedente

- Nel caso il nodo $n$ abbia un figlio destro, allora il successivo è il `min()` del sotto-albero destro

- Se invece $n$ non ha un figlio destro, allora il successivo è il primo padre per cui $n$ è nel sotto-albero sinistro

## Eliminazione

L'eliminazione nel caso di un nodo senza o con solo un figlio è di semplice implementazione (eliminazione diretta o sostituzione con il figlio) ma la casistica con doppio figlio richiede un algoritmo specifico:

1. Individuazione del nodo da eliminare $n$
2. Individuazione del successore $s$ di $n$
3. Se $s$ ha un figlio destro, lo si appende al padre di $s$ al posto di $s$
4. Infine si copia $s$ al posto di $n$

# Albero Red-Black

> ### Rotazioni
>
> Nel caso in cui un inserimento vada a rompere uno dei vincoli, si applicano le rotazioni:
> (caso di rotazione a sinistra)
>
> 1. Prendere il figlio sinistro del figlio destro di $n$ e metterlo come figlio destro
> 2. Mettere $n$ come figlio sinistro del suo figlio destro
> 3. Sostituire la vecchia posizione di $n$ rispetto al padre dall'ex figlio destro di $n$

**Inserimento**

> > 1. Memorizziamo il nodo coinvolto ($c$), suo padre($p$), suo zio($z$) e suo nonno ($n$)
> > 2. Valutiamo i 5 casi possibili:
> >    1. $c$ è il primo nodo
> >       1. lo inseriamo e lo coloriamo di nero
> >    2. $p$ è nero
> >       1. inseriamo $c$ colorato di rosso
> >    3. $c$, $p$ e $z$ sono rossi
> >       1. Coloriamo $p$ e $z$ di nero e $n$ di rosso
> >       2. Se $n$ è radice rossa o suo padre è rosso, aggiorniamo i nodi considerando $c = n$ e reiteriamo
> >    4. $c$ e $p$ rossi, e $z$ nero
> >       1. Con $c$ figlio **destro** di $p$ e $p$ figlio **sinistro** di $n$:
> >          - rotazione a **sinistra** da $p$
> >       2. Con $c$ figlio **sinistro** di $p$ e $p$ figlio **destro** di $n$:
> >          - Rotazione a **destra** da $p$
> >       3. proseguo nel caso 5
> >    5. $c$ e $p$ rossi, e $z$ nero
> >       1. Con $c$ figlio **sinistro** di $p$ e $p$ figlio **sinistro** di $n$:
> >          - rotazione a **destra** da $n$
> >       2. Con $c$ figlio **destro** di $p$ e $p$ figlio **destro** di $n$:
> >          - Rotazione a **sinistra** da $n$
> >       3. Coloro $n$ di rosso e $p$ di nero

# Priority queue

```
PriorityItem{
        int priority
        Item value
        int pos
}
```

Offre 4 funzioni:

- `insert()`: scorre il vettore fino alla posizione di priorità corretta, sposta tutti gli altri elementi
- `min()`: ritorna l'elemento in posizione 0
- `heapRestore()`:

  ```
  heapRestore(PriorityItem[] a, int i, int dim){
          int min = i
          if [2*i <= dim and a[2*i]<a[min]]{
                  min = 2*i
          }
          if [2*i+1 <= dim and a[2*i+1]<a[min]]{
                  min = 2*i+1
          }
          if [ i!=min ]{
                  swap(a,i,min)
                  heapRestore(a,min,dim)
          }
  }
  ```

- `deleteMin()`: elimina l'elemento in posizione 1 e chiama `heapRestore`
- `decrease(PriorityItem x, int p)`: sposta l'elemento fino a priorità `p`

# Grafi

> **Example**
>
> - `SET V()` (restituisce tutti i nodi)
> - `int size()`
> - `SET adj(Node u)`
> - `insertNode(Node u)`
> - `insertEdge(Node u, Node v)`
> - `deleteNode(Node u)`
> - `deleteEdge(Node u, Node v)`