PER CODICE DOCUMENTO CODIFICA CPP MONOKAI-SUBLIME: https://chrome.google.com/webstore/detail/code-blocks/ebieibfdjgmmimpldgengceekpfefmfd?utm_source=permalink

Duplicati
Realizzare una funzione Dup(A,p,r) che verifica, in modo divide et impera, la presenza di duplicati nell'array A.

```
//con array A ordinato
duplicati(int []A,int l,int r){
		if(l >= r)
				return false;
		else{
				c=(l+r)/2;
				return A[c]==A[c+1]|| duplicati(A,l,c) || duplicati(A,c+1,r); || duplicati(A,l,c) ||
duplicati(A,c+1,r);
		}
}

//con array A non ordinato
duplicati(int []A,int l,int r){
if(l >= r)
				return false;
else{
		c=(l+r)/2;//mi fermo appena trovo duplicato
		return duplicati(A,l,c)|| duplicati(A,c+1,r)|| merge_dup(A,l,c,r);
}
}

merge_dup(A,l,c,r){
//copio parte sinistra e parte destra come su merge
i=j=1;
for(k=l;k<r;k++)
		if(L[i]>R[j]){
				A[k]=R[j];
```

```
                j++;
        }
        elif(L[i]<R[j]){
                A[k]=L[i];
                i++;
        }
        else return true;
return false;//non erano presenti duplicati
}
```

Somma

Realizzare una funzione Sum(A,k) che dato un array A e un intero k verifica se k è esprimibile come la somma di due elementi di A ovvero se vi sono indici i e j tali che k = A[i] + A[j].

```
Sum(int []A,int k){
int n=A.length;
mergeSort(A,n);
for(int i=0;i<n;i++){
        bool cont=true;
        for(int j=n;cont&&0<j;j--){
                if(A[j]+A[i]==k)
                        return true;
                else if(A[j]+A[i]>k)
                        cont=false;
        }
}
}
```

Monete False

Siano date n monete, delle quali una è falsa. Le monete sono identiche esteticamente, ma quella falsa pesa meno delle altre. Data una bilancia a due piatti e avendo a disposizione come operazioni solo la pesata di due sottoinsiemi (disgiunti) di monete, dare un algoritmo di complessità O(log n) che determina qual è la moneta falsa. Mostrare che questa è la complessità del problema, ovvero che ogni algoritmo è Ω(log n).

```
MonFalse(int []A,int p,int u){
p1=(u-p)/3;
p2=p1*2;
if(Mon(A,0,p1) == Mon(A,p1,p2))
return di MonFalse(A,p2,n);
else if(Mon(A,0,p1) > Mon(A,p1,p2))
return MonFalse(A,0,p1);
else
return MonFalse(A,p1,p2);
}
```

Ha complessità di O(logn) con omega(logn

Risolvere le seguenti ricorrenze con Master Theorem:

- $T(n)=2T(n/2)+\log n$
- $T(n)=2T(n/2)+n^2$
- $T(n)=2T(n/2)+n\log n$

Tripartition furba

con elementi minori di x, uguali a x e quelli maggiori di x.

elem < x	elem = x	...elem da vedere	elem > x	elemento x

Heap (Max){//trova il massimo in un heap
}

Fusione di due alberi

```
Fusione(T1,T2){//unisce fondendo i due alberi T1 e T2 con la stessa altezza e completo T1
A = new int[T1.length*2];
A.heapsize = 1;
while(T1.heapsize > 0 && T2.heapsize > 0){
        mT1 = Max(T1);
        mT2 = Max(T2);
        if(mT1 > mT2)
                Insert(A,ExtractMax(T1));
        else
                Insert(A,ExtractMax(T2));

}
}
```

```
Fusione(T1,T2){//unisce fondendo i due alberi T1 e T2 con la stessa altezza e completo T1
        if(T1.root > T2.root){
                C=T2.root;
                T2.root = T1.root;
                T1.root = C;
                MaxHeapify(T1);
                r=Extract(T2);
                Create_Node(r);
        }
}
```

```
Fusione(T1,T2){//unisce fondendo i due alberi T1 e T2 con la stessa altezza e completo T1
	//si ragiona lavorando mettendo la radice massima di T1 nell'ultimo elemento di T2 e quell'elemento nella
	//radice di T1 quindi si esegue il MaxHeapifyUp su entrambi gli alberi e si continua così ricorsivamente
}
```

Inversione

Merge Sort in cui conto le inversioni

```
Inversione(A,p,r){//ritorna il numero di inversioni in A[p,r] con DIVIDE et IMPERA
if(p<r){
	c=p+r/2;
	return Inversione(A,p,c) + Inversione(A,c+1,r)
    + MergeConta(A,p,c,r);
    }
}

MergeConta(A,p,c,r){
    n1=c-p+1;
	L[]<A[p,c];
	R[]<A[c+1,r];
	int i=1, j=1, inv=0;
	for(int k=p; k < r; k++)
		if(L[i]>R[j]){
		A[k]=R[j];
		j++;
		inv+=inv+n1-i+1;//
		}
		else{
		A[k]=R[j];
		i++;
		}
return inv;
}
```

Selezione

elemento i da array non ordinato per posizione i ordinato

```
Select(A,i){//elemento in posizione i nell'array ordinato A[1...n] O(n+klog(n))

if(i> A.length/2){
	BuildMaxHeap(A);
```

```
	for(j=A.length down to k-1){//alla fine non sposto k così da lasciarlo nella posizione uno per return compatibilmente al minimo
		A[1]<->A[j];
		A.heapsize--;
		MaxHeapifyUp(A,1);
	}
}else{
	BuildMinHeap(A);
	for(i=1 up to k-1){
		ExtractMin(A);
	}
}
return A[1];
}
```

ora con complessità O(n*log(k))

```
Select(A,i){//con complessità O(n*log(k))
	BuildMaxHeap(A,k);//(k) costruisce solo prime k posizioni
	for(j=k+1 to n){//(n-k)
		if(A[j]<A[1]){
			A[1]<->A[j];
			MaxHeapify(A,1,k);//log(k)
		}
	}
	//O(k+(n-k)log(k)) ma siccome domina n togliendo k fuori da log rimane n*log(k)
return A[1];//alla fine i primi k elementi sono tutti più piccoli di quelli da j alla fine di A
}
```

Massimo con divide et Impera

Realizzare una procedure di tipo divide et impera Max(A,p,r) per trovare il massimo nell'array A[p,r]. Si assuma che l'array non sia vuoto (p d' r). Scrivere lo pseudocodice e valutare la complessit`a con il master theorem.

```
Max(A,p,r){
	if(p==r)
	return A[p]; //caso base
	else{
	m=(p+r)/2;
	m1=Max(A,p,m); //separazione parte uno
	m2=Max(A,m,r); //separazione parte due
	if(m1>=m2)	//combinazione
		return m1;
	else
		return m2;
	}
```

```
}
```

Ora valuto complessità con Master Theorem:

- elimino case base con m>1
- T(n)=T(n/2)+T(n/2) + c => T(n)=2T(n/2)+c
- applico teorema con n*log2(2) = n confronto questo con f(n)=c e ci troviamo nel primo caso siccome c/n all'infinito da 0 potendo concludere che quindi tempo stimato per l'algoritmo è O(n)

Range sottoalbero radice X

Implementare la funzione Range(x,k1,k2) che prende in input un nodo x di un albero binario di ricerca e due interi k1 e k2 e stampa, in ordine crescente di chiave, gli elementi nel sottoalbero con radice x, aventi chiave k tale che k1 <= k <= k2.

```
Range(x,k1,k2){
        if(k1 <= x.key <= k2){
                Range(left(x),k1,k2);
                print x;
                Range(right(x),k1,k2);
        }else if(k1 >= x.key)
                Range(right(x),k1,k2);
        else
                Range(left(x),k1,k2);

}
```

Insert con predecessore

Si supponga che gli alberi binari di ricerca abbiano nodi che oltre al padre e a i figli, memorizzino anche il predecessore in un campo prec. Realizzare la procedura Insert(T,z) di modo che memorizzi correttamente anche precedecessore del nodo.

```
Insert(int T,int z){
        x=T.root;

        T.heapsize++;
        T[T.heapsize].key=z;
        T[T.heapsize].father=T[T.heapsize/2];
}
```

Risoluzione compitino

Esercizio 1:

n^2(n+1)

n^2(n+1)= Ω(n^2+eps) con 0<eps<1

T(n)=Θ(f(n))=

condizione di regolarità: a*f(n/b) <= k * f(n) con 0<k<1

4*(n/2)^2*(n/2 + 1)= (n^2(n+2))/ 2 <= kn^2(n+1)

k>= n+2/2(n+1) quindi per ogni n>=1 con k=¾ la condizione è vera
Se all'inizio volevo mettere uguale a Θ(n^3) il membro

Esercizio 2:
numero di tentativi parte da zero.
h1=k mod 8 k2=12 *(k mod 7)
h(k,i)=(h1(k)+i*h2(k))mod m
34→h1(34)=2
12→h1(12)=4
18→h1(18)=2 → h(18,1) → h1(18)+ 1* h2(18)
23→ "7"
15→ "5"

InsertABR

con campo f indicante il numero di foglie nel sottoalbero radicato, x.f.
Scriverne l'algoritmo di

```
Insert(T,z){
      x=T.root;
      y=nil;//lavoro doppio con y genitore e x figlio mi fermo con x ma inserisco su y
      while(y != T.nil){
            if(x.key > z.key)
                  x=x.left;
            else
                  y=y.right;
      }
      z.parent=y;
      z.f=1;
      if(y==nil)
            y=z;//radice albero
      else
            if(z.key < y.key)
                  y.left=z;
            else
                  y.right=z;

      if(y.left && y.right ){//y non è una foglia
            //aumento tutti campi f albero
            y.f=y.f+1;
            y=y.p;
      }
}
```

Esercizi ricorrenze con masther theorem e alberi con sostituzione

$T(n)=T(n/2)+c$
nlogb^a=n^0=1 -> O(1)
f(n)=c=O(1) -> master theorem O(1*log n)

$T(n)=T(n/2) + T(n/4) + n$ → per $T(n/4)$ non vale il teorema dell'esperto, facciamo quindi albero per ipotesi di costo che troviamo nel livello 1 pari a n, 2 a ¾ n e sul $(¾)^2$ n quindi stimo costo di O(n) .
Devo quindi dimostrare che $T(n)=\Omega(n)$ ossia esiste un c>0 tale che T(n)>= c*n per ogni n e inoltre che T(n)=O(n) ossia esiste un d>0 tale che T(n)=< d*n per ogni n.
Lo possiamo dimostrare tramite prova induttiva ossia
T(n) = T(n/2) + T(n/4) + n e per ipotesi induttiva
T(n) >= c*n/2 + c*n/4 + n → T(n) >= c*n ¾ + n e pongo questo maggiore di >= cn quindi ¼ cn <= n diventa c <= 4 ossia esiste un c>0 con le proprietà desiderate.

Ricerca aule libere minimizzando tempo sprecato

date lezioni ordinati per tempo finale crescente scelta greedy allocando L1 in Ak con k minimo indice ak <= s1 e differenza s1-ak minimo, esiste una allocazione ottima con questa scelta.
Se nessuna è libera allora alloco una nuova aula. Complessità O(n^2) ma può diventare O(n*logn)

```
ProgAttivita(s,t,n){
  numAule=0;
  for(i=1 to n){
    j=1;
    min=+infinito;
    for(j=1 to numAule){
      if(s[i]>q[j])
        if(s[i]-a[j]<min){
          min=s[i]-a[j];
          alloc[i]=j;
        }
        if(min==+infinito){
          numAule++;
          alloc[i]=numAul;
        }//if +inf
        a[alloc[i]]=t[i];
    }//for j
  }//for i
  return numAule
}
```

LIS - Longest increasing subsequences

X=x1...xn sottosequenza di x crescente di lunghezza massima
li=lunghezza di una LIS che inizia in xi, sia Y=y1...yk è LIS di x se e solo se:

1. se x1!=y1 allora y è LIS di x2...xn
2. se x1=y1 allora y2...yk è LIS x2...xn dove elimino gli xj < xi

li{1 se i=n altrimenti 1+max i+1<=j<=n xi<=xj (lj) se i < n
Ora procedimento analogo a tutti esercizi programmazione dinamica, 1. corsivo e poi 2. LIS(x,n) algoritmo con normalizzazione

```
LIS(x,n){
  X[0]=-infinito;
  for(i=0 to n){
    L[i]=-infinito;
  }
  return LIS_rec(X,n,L,0);
}

LIS_rec(X,n,L,i){
  if(L[i]==-infinito)
    if(i==n)
      L[i]=1;
    else{
      max=0;
      for(j=i+1 to n){
        q=LIS_rec(x,n,L,j);
        if(X[i]<= X[j] && max<q)
          max=L[j];
      }
      L[i]=max+1;
    }
    return L[i];
}

print(X,N){
  i=0;
  while(N[i]!=NIL){
    i=N[i];
    print X[i];
  }
}

print_rec(X,N,i){
```

```
  print X[i];
  if(N[i] != NIL)
    print_rec(X,N,N[i]);
}
```

Incremento e Reset AMMORTIZZATO

```
Reset(A){
  for(i=0 to last){
    A[i]=0;
  }
}

Inc(A){
  i=0;
  while(i < k && A[i]==1){
    A[i]=0;
    i++;
  }
  if(i<k){
    A[i]=1;
    A.last=max{i,A.last};
  }
}
```

Interval Partitioning Problem
Date n lezioni A1, ..., An, ciascuna con il suo tempo di inizio si e fine fi scrivere un algoritmo per allocare tutte le lezioni in un numero minimo di aule.
Problema risolvibile con algoritmo greedy ordinando le richieste da quella con tempo iniziale minore, nel set rimanente di lezioni sceglieremo sempre la prima con il tempo d’inizio minore di tutte le altre o in caso alternativo chiederemo una nuova risorsa(aula da utilizzare)

```
AuleLezioniMinime(A,s,f){
  Aula d *=[];i=0;
  while(!A.empty()){
    i=minAtt(A,s);
    if(j = AssignAula(i,d)){
    d(j).push(A.remove(i));
    }else{
    d=[d; new Aula()];
    d(d.length).push(A.remove(i));
```

```
    }
  }
}
```

Definiamo A come il massimo numero di risorse in conflitto in ogni momento in A, è chiaro che ogni set di richieste A, il numero di richieste è almeno la profondità di A perchè le lezioni in conflitto devono sempre essere assegnate ad aule diverse. Dimostriamo quindi per contraddizione che situazione con aule A e n pari al numero di Aule in A vogliamo dimostrare che l'algoritmo greedy almeno n+1 risorse il che vorrà dire avendo Aule ordinate in senso crescente per orario d'inizio quindi per la risorsa j avremo almeno almeno n risorse in conflitto ma siccome nel nostro caso abbiamo n+1 di profondità questo contraddice l'ipotesi dimostrando che l'uso di n risorse è ottimale.8

Esercizio File system

File disco
file dim f1,f2,...,fn
capacità d massima del disco
Obiettivo: massimizzare il numero di file memorizzati
Quindi assumendo f1 <= f2 <= ... <= fn quindi ordinati in modo crescente

```
Files(f,d,n){
  I=0;
  cap=d;
  i=1;
  while(i <= n && cap >= fi){
    I=I u {i};
    cap= cap - fi;
    i++;
  }
}
```

Ora dobbiamo però dimostrare due cose:
1. sottostruttura ottima
dato insieme file F={f1,...,fn} e capacità d
con una soluzione **I** c {1,...,n} -> ottima

Vogliamo dire che **I** è fatta con una sottoscelta ottima del sottoproblema.
Quindi **I**!=0
w(**I**)=Sommatoria di fi <= d
c(**I**)=|**I**| massimizzazione

Dato k in **I** allora F'=F\{k} quindi d'=d-k allora **I**'=**I**\{k} è soluzione ottima.

Dato k in I allora F'=F\{k} quindi d'=d-k allora I'=I\{k} è soluzione ottima.
I' è soluzione per sottostruttura quindi
W(I')=Sommatoria fi = Sommatoria fi-fk <= d-fk
= W(I) <= d
I' è ottima per sottostruttura sia I' non ottima per sottostruttura.
=> c'è I'' c {1,...,k-1,k+1,...,n} soluzione
tale che |I''|>|I'| allora I''u{k} è soluzione per problema originale "migliore" di I
w(I''u{k}) <= d
|I''u{k}|>|I|

2. scelta greedy

n(i,d)= costo(numero di file) di soluzione ottima per {f1,...,fn} con spazio d
n(i,d)={ 0 se i=0
n(i-1,d) se i>0 e fi > d
max{n(i-1,d),n(i-1,d-fu)+1} se i>0 e fi <= d
}
Allora esiste ottima I c {1,...,n} tale che 1 in I
dim. sia I' una qualunque soluzione ottima e sia k in I'
allora I'\{k}u{1} è ottima

Resto r con tagli 5,2,1 siccome sono limitati è possibile la scelta greedy

ABR con attributo addizionale x.f = numero di foglie sottoalbero

```
Insert(T,z){
  x=T.root;
  y=nil;
  while(x!=nil){
  y=x;
  if(z.key < x.key)
    x=x.left;
  else{
    y=y.left;
    z.p=y;
    if(y==nil)
      T.root=z;
    else{
      if(z.key < y.key)
        y.left=z;
      else
        y.right=z;
    }
```

```
    z.f=1;
  }
  }
  if(y.left != nil && y.right != nil){
    while(y!=nil){
      y.f=y.f+1;
      y=y.p;
    }
  }
}
```

Esercizio hash aperto:
m=9
h1(k)=k mod n
h2(k)=1 + k mod(m-2)

h(k,i)

T(n)=T(n-2) + n^3
O(n^4) → metodo di sostituzione
T(n) <= c*n con c>0
T(n) <= c(n-2)^4 + n^3
T(n) <= c*n^3 (n-2)+n^3

T albero buono e verificare se è completo

```
Complete(x){// --> true/false e altezza
  if(x==nil){
    return true;
  }else{
    return (complete(x)&& complete(y));
  }
}

complete(x){
  if(x==nil){
    return true;
  }else{
    cleft, hleft = complete(x.left);
    cright, hright = complete(x.right);
    c= c.left and c.right and (hleft == hright);
    h= max{hleft,hright}+1;
```

```
  }
  return c,h;
}
```

ISMAXHEAP

```
IsMaxHeap(A,n,i){
  if(2*i+2 > n)
    return true;

  if(A[i]>A[2*i] && A[i]>A[2*i+1])
    return (IsMaxHeap(A,n,i*2) && IsMaxHeap(A,n,i*2));
  else
    return false;
}
```

ISPRIORITYQUE - MinHeap

```
IsPriorityQue/IsMinHeap(A,n,i){
  if(2*i+2 > n)
    return true;

  if(A[i]<A[2*i] && A[i]<A[2*i+1])
    return (IsMinHeap(A,n,i*2) && IsMinHeap(A,n,i*2));
  else
    return false;
}
```

ISABR - albero ricerca binaria

```
ISABR(A,n,i){
  if(i>n) return true;
  else{
    minL=MAX(A,n,2*i);
    maxR=MIN(A,n,2*i+1);
    okl=ISABR(A,n,2*i);
    okr=ISABR(A,n,2*i+1);
    return okl && okr && maxL <= A[i] && A[i] <= minR;
  }
}
```

ISARN - albero rosso nero

```
IsARN(A, n, i) {
    if (A == NULL) return 1;
    int leftBlackHeight = IsARN(A[i]->left);
    if (leftBlackHeight == 0)
      return leftBlackHeight;
    int rightBlackHeight = IsARN(A[i]->right);
    if (rightBlackHeight == 0)
      return rightBlackHeight;
    if (leftBlackHeight != rightBlackHeight)
      return 0;//altezza diversa quindi non rispetta proprietà
    else
      return leftBlackHeight + A[i]->IsBlack() ? 1 : 0;
}
```

SDEGREE

Fornire lo pseudocodice di una funzione sdegree(T) che calcola il grado di squilibrio dell'albero T (si possono utilizzare funzioni ricorsive di supporto).

```
sdegree(T){
if(T.left == NULL && T.right == NULL) return 0;
 i=sdegree_ric(T.left)-sdegree_ric(T.right);
 l=sdegree(T.left);
 r=sdegree(T.right);
 max=i;
if(l>i)
  max=l;
if(r>i)
  max=r;
return max;
}
sdegree_ric(T){
if(T.key == NULL) return 0;
  return T.key+(sdegree_ric(T.left) + sdegree_ric(T.right));
}
```

BANCONOTE MINIMO N

Si supponga di dover pagare una certa somma s. Per farlo si hanno a disposizione le banconote b1; : : : ; bn ciascuna di valore v1; : : : ; vn. Si vuole determinare, se esiste, un insieme di banconote bi1 ; : : : ; bik che totalizzi esattamente la somma richiesta e che minimizzi il numero k di banconote utilizzate.

```
banconote(S,V,n){
  for(j=0 to n){
    c[S',j]=-1;
  }
  return banconote_ric(S,v,n,c);
}
banconote_ric(S,v,j,c){
  if(c[S',j]==-1){
    if(S'==0)
      C[S',j]=0;
    else if(j==0)
      C[S',j]=-infinito;
    else  if(j > 0 && v[j] <= S')

C[S',j]=min{banconote_ric(S'-v[j],j-1),banconote_ric(S',j-1)}
    else
      C[S',j]=banconote_ric(S',j-1);
  }
  return C[S',j];
}
```

MassimoPalindrome

```
MaxPalindroma(S){
  n=S.length;
  for(i=1 to n){
    l[i,i-1]=0;
    l[i,i]=0;
    b[i,i]=i;
  }
  for(len = 1 to n)
    for(i=1 to n-len+1){ //=(n^2 con ciclo len e i)
      if(S[i]==S[j] && l[i+1,j-1]==j-i+1){
        l[i,j]=l[i+1,j-1]+2; b[i,j]=i;
      }
      else{
        if(l[i+1,j]>=l[i,j-1]){
          l[i,j]=l[i+1,j];
          b[i,j]=b[i+1,j];
        }
```

```
        else{
          l[i,j]=l[i,j-1];
          b[i,j]=b[i,j-1];
        }
      }
  return l[1,n];
}
```

MaxPath

```
MaxPath(T){
if(T.root == NULL)
  return 0;
else if(T.left == NULL && T.right == NULL)
  return 0;
else
  return max{MaxPathRic(T.left),MaxPathRic(T.right)};
}

MaxPathRic(x){
if(x == NULL)
  return 0;
else if(T.left == NULL && T.right == NULL)
  return x.key;
else
  return x.key+max{MaxPathRic(x.left),MaxPathRic(x.right)};
}
```

Struttura dati OrderedStack

OEmpty(S)
OPop(S)
OTop(S)
OPush(S,x)