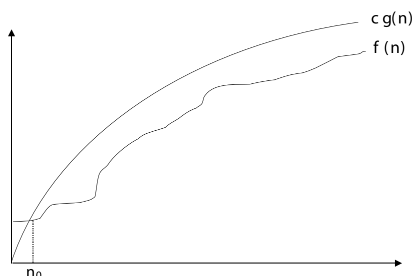
Lezione 1 (10.03.2010)

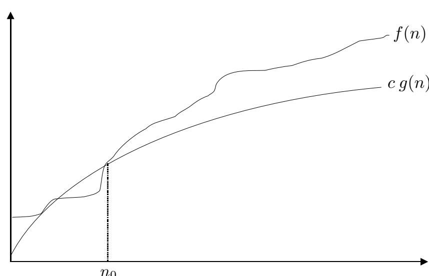
Notazione asintotica O

$f(n) = O(g(n))$ se $\exists\, c > 0$ e $n_0 \geq 0$ tali che $f(n) \leq c\, g(n)$ per ogni $n \geq n_0$



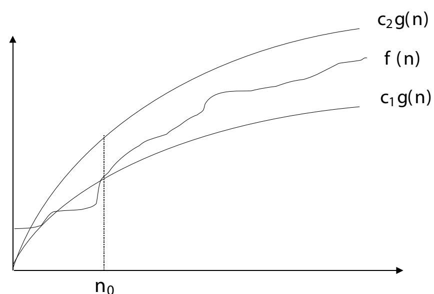
Notazione asintotica Ω

$f(n) = \Omega(g(n))$ se $\exists\, c > 0$ e $n_0 \geq 0$ tali che $f(n) \geq c\, g(n)$ per ogni $n \geq n_0$



Notazione asintotica Θ

$f(n) = \Theta(g(n))$ se $\exists\, c_1, c_2 > 0$ e $n_0 \geq 0$ tali che $c_1 g(n) \leq f(n) \leq c_2 g(n)$ per ogni $n \geq n_0$



Qualche utile regoletta

$f(n) = O(g(n)) \leftrightarrow g(n) = \Omega(f(n))$

$f(n) = O(g(n)) \leftrightarrow a\,f(n) = O(g(n)),\ \forall\, a > 0$

$f(n) = O(g1(n)),\ f_2(n) \leftrightarrow O(g_2(n)) \rightarrow f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$

$f(n) = O(g1(n)),\ f_2(n) \leftrightarrow O(g_2(n)) \rightarrow f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$

$f(n) = \Theta(g(n)) \leftrightarrow g(n) = \Theta(f(n))$
$f(n) = \Theta(g(n)) \rightarrow f(n) = O(g(n)) \rightarrow g(n) = \Omega(f(n))$
$f(n) = \Theta(g(n)) \rightarrow f(n) = \Omega(g(n)) \rightarrow g(n) = O(f(n))$
$f(n) = O(g(n) \wedge g(n)) = \Omega(h(n)) \rightarrow f(n) = O(h(n))$

per qualsiasi $r < s$, $h < k$, $1 < a < b$:
$O(1) \subset O(\log^r n) \subset O(\log^2 n) \subset O(n^h) \subset O(n^h \log^r n) \subset O(n^h \log^s n) \subset O(n^k) \subset O(a^n) \subset O(b^n) \subset O(n^n)$

Esercizio 1

```
int F (int v[], int n)
   //v: vettore, n: dimensione del vettore v
   int s;
   s = v[0] + v[n − 1];
   return s;
```

Soluzione:

complessità costante $T(n) \in O(1)$

Esercizio 2

```
int F (int v[], int n)
   //v: vettore, n: dimensione del vettore v
   int s;
   s = v[0];
   for (int i = 1; i < n; i + +) do
      s+ = v[i];
   return s;
```

Soluzione:

$$T(n) = c_1 + c_2 + \sum_{i=1}^{n-1} c_3 = c_1 + c_2 + (n-1)c_3 = n(c_2 + c_3) + c_1 - c_3 \in O(n)$$

Esercizio 3

```
int F (int v[], int n)
    //v: vettore, n: dimensione del vettore v int s;
    s = v[0];
    for (int i = 1; i < n; i++) do
        s+ = C(v, n);
    return s;
```

```
int C(v, n)
    int s;
    s = v[0];
    for (int i = 1; i < n; i++) do
        s+ = v[i];
    return s;
```

Soluzione:

$$T(n) = c_1 + nc_2 + \sum_{i=1}^{n-1}(c_3 + T_G(n))$$

$$T_G(n) = c_4 + nc_5 + \sum_{i=1}^{n-1}(c_6) = c_4 + nc_5 + (n-1)c_6 = n(c_5 + c_6)(c_4 - c_6) = O(n)$$

$$T(n) = c_1 + nc_2 + (n-1)c_3 + (n-1)O(n) = (c_1 - c_3)O(n) + n(c_2 - c_3) + nO(n) \in O(n^2)$$

Esercizio 4

```
int F (int v[], int n)
    int s = v[0], i = 0;
    while (i < n) do
        s+ = v[i];
        i+ = 2;
    return s;
```

Soluzione:

$$T(n) = \sum_{i=0}^{(n/2)-1}(c_1 + c_2) \in O\left(\frac{n}{2}\right) \in O(n)$$

Esercizio 5

```
int F (int v[], int n)
    int s;
    s = v[0];
    for (int i = 1; i < n; i++) do
        for (int j = 1; j < i; j++) do
            s+ = v[j];
    return s;
```

Soluzione:

$$T(n) = \sum_{i=1}^{n-1}\left(\sum_{j=1}^{n-1} c_1\right) = \sum_{i=1}^{n-1}((n-1)c_1) = c_1\sum_{i=1}^{n-1}(n-1) = c_1\frac{n(n-1)}{2} \in O\left(n^2\right)$$

Esercizio 6

```
int F (int n)
-----------------------------
int m, x;
if (n>2) then
    x = n;
    m = n;
    while (m > 1) do
        x = x * m;
        m = m - 2;
    return (x + F(n/2));
return 0;
```

Soluzione:

$$T(n) = \alpha + \beta\frac{n}{2} + T\left(\frac{n}{2}\right) = 2\alpha + \beta\frac{n}{2} + \beta\frac{n}{4} + T\left(\frac{n}{4}\right) = 3\alpha + \beta\frac{n}{2} + \beta\frac{n}{4} + \beta\frac{n}{8} + T\left(\frac{n}{8}\right)$$

$$n = 2^h \qquad \text{dopo } h-1 \text{ passi}$$

$$(h-1)\alpha + \beta\sum_{i=1}^{n-1}\frac{n}{2^i} + T(2)$$

$$\alpha\log\frac{n}{2} + \beta\sum_{i=1}^{n-1}\frac{n}{2^i} + O(1)$$

$$\alpha\log\frac{n}{2} + n\beta\cdot\frac{1}{2}\cdot\frac{1-\frac{1}{2^{h-1}}}{1-\frac{1}{2}}$$

$$\alpha\log\frac{n}{2} + n\beta\cdot\frac{1}{2}\cdot\left(1-\frac{1}{2^{h-1}}\right)$$

$$\alpha\log\frac{n}{2} + n\beta\cdot\frac{1}{2}\cdot\left(1-\frac{2}{n}\right)$$

$$\left(\alpha\log\frac{n}{2} + n\beta - 2\beta\right) \in O(n)$$

$$T(n) = \begin{cases} c & \text{se } n \leq 2 \\ (c_1+c_2) + \frac{n}{2}(c_3+c_4) + T(\frac{n}{2}) & \text{se } n > 2 \end{cases}$$

Esercizio 7

```
int F (n)
if n == 1 then
  return 1;
else
  return F(n − 1) + F(n − 1);
```

Soluzione:

$$T(n) = \begin{cases} O(1) & \text{se } n = 1 \\ c + 2T(n-1) & \text{se } n \geq 1 \end{cases}$$

Esercizio 8

```
int F (n)
if n == 0 then
  return 0;
else if n%3 == 1 then
  return 1;
else if n%3 == 2 then
  return 2;
else
  return F(n/3) + F(n/3) + F(n/3);
```

Soluzione:

$$T(n) = \begin{cases} \alpha & \text{se } n = 0,\ n\%3 = 1,\ n\%3 = 2 \\ 3T(n/3) + \beta & \textit{altrimenti} \end{cases}$$

Lezione 2 (24.03.2010)

Esercizio 9

```
int foo (int v[], int i, int j, int k)
    int m = (i + j)/2;
    if (v[m] == k) then
        return 1;
    else
        if (v[m] > k) then
            return foo(v, m + 1, j, k)
        else
            return foo(v, i, m − 1, k)
```

Soluzione:

$$T(n) = \begin{cases} \alpha & \text{se } v[m] = k \\ T\left(\frac{n}{2}\right) + \gamma & \text{se } v[m] \neq k \end{cases}$$

$$T(n) = T\left(\frac{n}{2}\right) + \gamma = T\left(\frac{n}{2^2}\right) + 2\gamma = T\left(\frac{n}{2^3}\right) + 3\gamma = T\left(\frac{n}{2^4}\right) + 4\gamma$$

$$\text{dopo } h \text{ passi} \quad T(n) = T\left(\frac{n}{2^h}\right) + h\gamma$$

$$\text{supponendo } n = 2^h \quad T(n) = \alpha + \gamma \log n \in O(\log n)$$

Alberi di ricorsione

Il metodo iterativo diventa difficile in casi come questo:

$$T(n) = T\left(\frac{n}{3}\right) + T\left(2\frac{n}{3}\right) + n$$

$$\text{livello 2} = \begin{cases} T(n) = T\left(\frac{n}{9}\right) + T\left(2\frac{n}{9}\right) + \frac{n}{3} + T\left(2\frac{n}{9}\right) + T\left(4\frac{n}{9}\right) + 2\frac{n}{3} + n \\ = T\left(\frac{n}{9}\right) + 2T\left(2\frac{n}{9}\right) + T\left(4\frac{n}{9}\right) + 2n \end{cases}$$

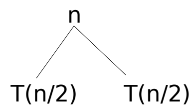
$$\text{livello 3} = \begin{cases} T(n) = T\left(\frac{n}{27}\right) + T\left(2\frac{n}{27}\right) + \frac{n}{9} + 2T\left(2\frac{n}{27}\right) + 2T\left(4\frac{n}{27}\right) + 4\frac{n}{9} + T\left(4\frac{n}{27}\right) + T\left(8\frac{n}{27}\right) + 4\frac{n}{9} + 2n \\ = T\left(\frac{n}{27}\right) + 3T\left(2\frac{n}{27}\right) + 3T\left(4\frac{n}{27}\right) + T\left(8\frac{n}{27}\right) + 3n \end{cases}$$

Una variante del metodo iteratico che si può usare in questi casi è il metodo dell'alvero di ricorsione

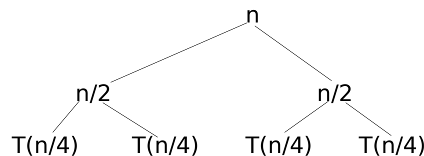
Alberi di ricorsione: $T(n) = 2T\left(\frac{n}{2}\right) + n$

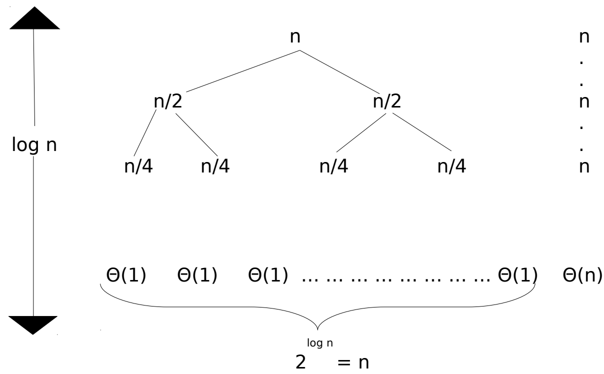
Espandiamo l'albero di ricorsione per passi

1° Passo



2°Passo



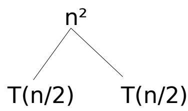


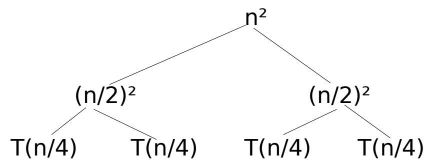
Complessivamente:

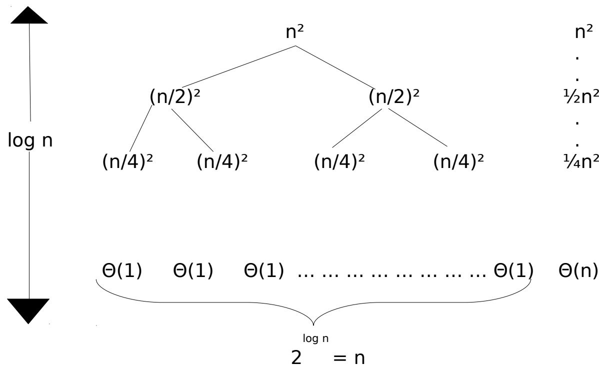
$$T(n) = \Theta(n) + n \cdot \log n = \Theta(n \log n)$$

Alberi di ricorsione: $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

Espandiamo l'albero di ricorsione per passi







Complessivamente:

$$T(n) = \Theta(n) + n^2 \cdot \sum_{i=0}^{\log n - 1} \frac{1}{2^i}$$

Svolgiamo la sommatoria:

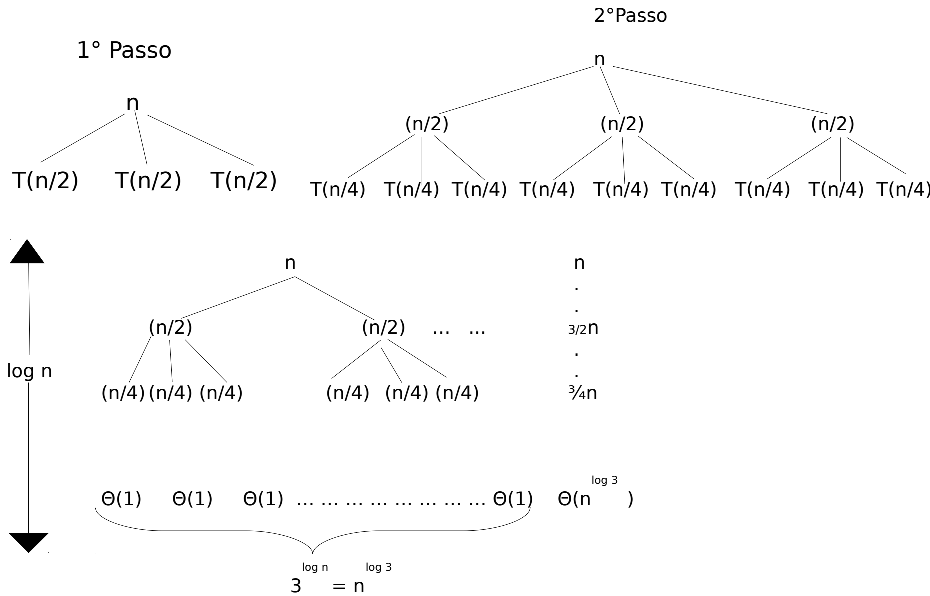
$$\sum_{i=0}^{\log n-1} \frac{1}{2^i} \leq \sum_{i=0}^{\infty} \frac{1}{2^i} = \frac{1}{1-\frac{1}{2}} = 2$$

Quindi

$$T(n) = \Theta(n) + 2n^2 = \Theta\left(n^2\right)$$

Alberi di ricorsione: $T(n) = 3T\left(\frac{n}{2}\right) + n$

Espandiamo l'albero di ricorsione per passi



Complessivamente

$$T(n) = \Theta(n^{\log 3}) + n \cdot \sum_{i=0}^{\log n-1} \left(\frac{3}{2}\right)^i$$

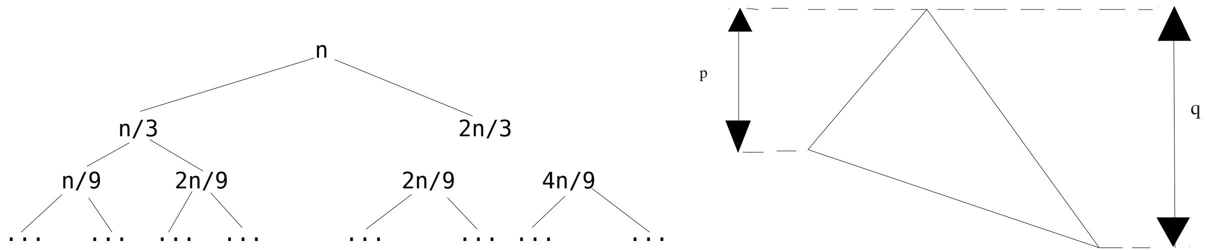
Svolgiamo la sommatoria:

$$\sum_{i=0}^{\log n-1} \left(\frac{3}{2}\right)^i = \frac{\left(\frac{3}{2}\right)^{\log n} - 1}{\frac{3}{2} - 1} = 2\left(\left(\frac{3}{2}\right)^{\log n} - 1\right) = 2\left(n^{\log \frac{3}{2}} - 1\right) \approx n^{\log \frac{3}{2}} = n^{\log 3} - n^{\log 2} = n^{\log 3 - 1} = \frac{n^{\log 3}}{n}$$

Quindi

$$T(n) = \Theta(n^{\log 3}) + n \cdot \frac{n^{\log 3}}{n} = \Theta(n^{\log 3}) + n^{\log 3} = \Theta(n^{\log 3})$$

Alberi di ricorsione: $T(n) = T\left(\frac{n}{3}\right) + T\left(2\frac{n}{3}\right) + n$



Somma dei nodi a livello $i \to n$
Somma dei nodi dell'albero di altezza $p = \log_3 n$:

$$\sum_{i=0}^{p-1} n = n \log_3 n = \Theta(n \log n) = \Omega(n \log n)$$

Somma dei nodi dell'albero di altezza $q = log_{\frac{3}{2}} n$:

$$\sum_{i=0}^{q-1} n = n \log_{\frac{3}{2}} n = \Theta(n \log n) = O(n \log n)$$

$$\Rightarrow T(n) = \Theta(n \log n)$$

Lezione 3 (25.03.2010)

Master Theorem

Siano a e b costanti intere tali che $a \geq 1$ e $b \geq 2$, e c, d e β costanti reali tali che $c > 0$, $d \geq 0$, e $\beta \geq 0$. Sia $T(n)$ data dalla relazione di ricorrenza:

$$\begin{array}{ll} T(n) = d & \text{per } n = 1 \\ T(n) = aT\left(\frac{n}{b}\right) + cn^{\beta} & \text{per } n > 1 \end{array}$$

Posto $\alpha = \frac{\log a}{\log b}$, allora:

$$\begin{array}{ll} (1)\, T(n) \text{ è } O(n^{\alpha}) & \text{se } \alpha > \beta \\ (2)\, T(n) \text{ è } O(n^{\alpha} \log n) & \text{se } \alpha = \beta \\ (3)\, T(n) \text{ è } O(n^{\beta}) & \text{se } \alpha < \beta \end{array}$$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \qquad \text{con } a \geq 1,\ b > 1$$

1. $f(n) = O(n^{\log_b a - \epsilon})$ per $\epsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = O(n^{\log_b a})$ $\Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b a + \epsilon})$ per $\epsilon > 0$, $\exists c < 1,\ n_0 \geq 0 |\, \forall n \geq n_0,\ af(\frac{n}{b}) \leq cf(n) \Rightarrow T(n) = \Theta(f(n))$

Esercizi

$T(n) = \sqrt{2}T\left(\frac{n}{2}\right) + \log n$
sol. $f(n) = \log n = O(\sqrt{n}) = O(n^{\frac{1}{2} - \frac{1}{3}})$ $\quad T(n) = \Theta(\sqrt{n})$

$T(n) = 3T\left(\frac{n}{3}\right) + \frac{n}{2}$
sol. $f(n) = \frac{n}{2} = \Theta(n)$ $\quad T(n) = \Theta(n \log n)$

$T(n) = 3T\left(\frac{n}{2}\right) + n^2$
sol. $f(n) = n^2 = \Omega(n^{\log 3 + 0.2})$ $\quad 3\left(\frac{n^2}{4}\right) \leq \left(\frac{4}{5}\right) n^2$ $\quad T(n) = \Theta\left(n^2\right)$

$T(n) = 2T\left(\frac{n}{2}\right) + n \log n$
sol. $f(n) = n \log n = \Omega(n)$ $\quad \log n = o(n^{\epsilon}) \Rightarrow n \log n = o\left(n^{1+\epsilon}\right)$

$T_1(n) = 4T\left(\frac{n}{2}\right) + n$

$T_2(n) = 4T\left(\frac{n}{2}\right) + n^2$

$T_3(n) = 4T\left(\frac{n}{2}\right) + n^3$

Proviamo a "interpretare" le complessità ottenute

Caso 1.

$$T_1(n) = 4T\left(\frac{n}{2}\right) + n = \Theta(n^2)$$

si applica se il costo della divisione in sottoproblemi e della costruzione della soluzione a partire da quella dei sottoproblemi (n^c) è trascurabile rispetto al costo della soluzione dei sottoproblemi stessi (a $T\left(\frac{n}{b}\right)$)

Caso 2.

$$T_2(n) = 4T\left(\frac{n}{2}\right) + n^2 = \Theta(n^2 \log n)$$

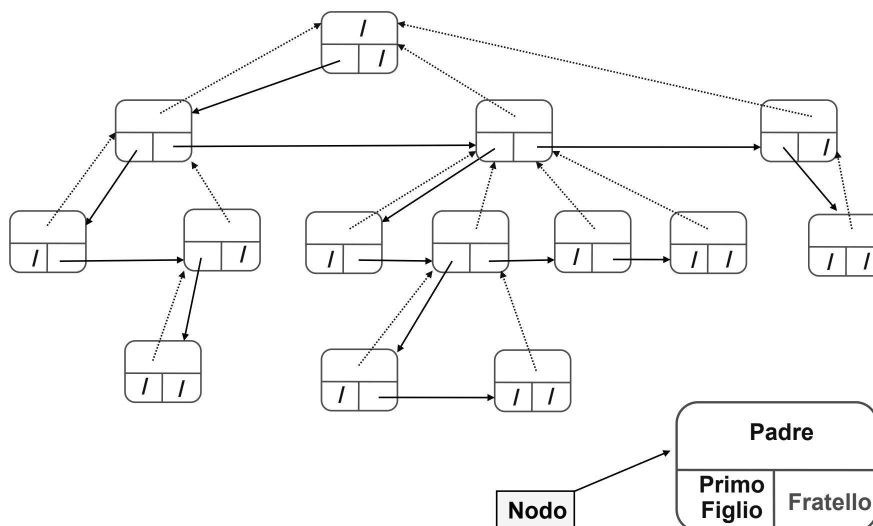
si applica se il costo della divisione in sottoproblemi e della costruzione della soluzione a partire da quella dei sottoproblemi (n^c) è analogo al costo della soluzione dei sottoproblemi stessi (a $T\left(\frac{n}{b}\right)$)

Caso 3.

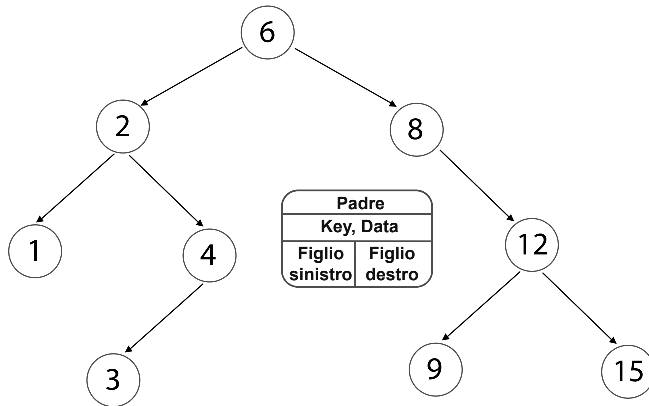
$$T_3(n) = 4T\left(\frac{n}{2}\right) + n^3 = \Theta(n^3)$$

si applica se il costo della divisione in sottoproblemi della costruzione della soluzione a partire da quella dei sottoproblemi (n^c) è il fattore dominante

Rappresentazione albero



Rappresentazione BST



RaddoppiaDisp

Dato un albero A, scrivere un algoritmo che trasforma A raddoppiando i valori di tutte le chiavi sui livelli dispari dell'albero. Si supponga che $root[A]$ sia a livello 0.
Soluzione:

```
RaddoppiaDisp (Node x, bool raddoppia)
if x != NIL then
    if raddoppia then
        chiave[x] ← 2 * chiave[x]
RaddoppiaDisp(fratello[x], raddoppia)
RaddoppiaDisp(figlio[x], !(raddoppia))
```

ContaMin

Sia T un albero binario di ricerca che implementa un dizionario. Sia v un nodo interno a T, e sia T_v il sottoalbero con radice v. Si proggetti un algoritmo efficiente che, ricevuto in input un nodo $v \in T$ e una chiave k, restituisce il nuero di entry in T_v con chiave $\leq k$.
Soluzione:

```
ContaMin (v,k)
if v = NIL then
    return 0
q ← v.element().key()
if q > k then
    return ContaMin(left(v), k)
else
    return 1 + ContaMin(left(v), k) + ContaMin(right(v), k)
```

TestMax

Dato un albero A si sviluppi un algoritmo per verificare se è soddisfatta la seguente proprietà speciale: $key\,[padre\,[x]] > key\,[x]$, per ogni x diverso dalla radice

Soluzione:

```
Test(A)
---------------------------------------------
x ← root[A];
return TestMax(x)
```

```
TestMax (x)
---------------------------------------------
if ((padre[x] = NIL) or (key[padre[x]] > key[x])) then
  return (TestMax(figlio[x]) and TestMax(fratello[x]))
else
  return false
```

Lezione 4 (29.03.2010)

UnionPrint

Siano A un albero binario di ricerca contenente n interi e B un array ordinato contenente m interi.
(a) Scrivere lo pseudo codice di una procedura efficiente per stampare in ordine non decrescente tutti gli elementi (merging) di A e di B.
(b) Indicare la complessità della procedura in funzione di n ed m.
Soluzione:

```
UnionPrint (BST A, vect B)
----------------------------------------
x ← minimum(A);
i ← 1;
while (x! = NIL) and (i ≤ m) do
    if (key[x] ≤ B[i]) then
        stampa key[x];
        x ← succ[x];
    else
        stampaB[i];
        i ← i + 1;
while (x! = NIL) do
    stampa key[x];
    x ← succ[x];
while i ≤ m do
    stampa B[i];
    i ← i + 1;
```

Contagrado

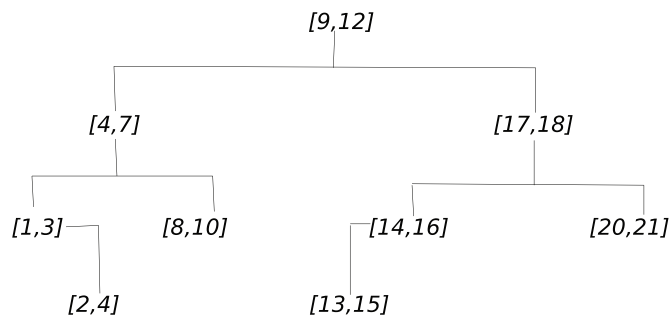
Si sviluppi un algoritmo che, dati un albero T e un intero positivo $k > 0$, conta il numero di nodi di grado k in T.
(Si supponga che l'albero sia rappresentato tramite gli attrubuti: fratello e figlio. Il grado di un nodo di un albero è pari al numero dei suoi figli).
Soluzione:

```
ContaGrado (x,k)

if x = NIL then
  return 0
z ← figlio[x]
n ← 0
while z! = NIL do
  n ← n + 1
  z ← fratello[z]
if n = k then
  return 1 + ContaGrado(fratello[x], k) + ContaGrado(figlio[x], k)
else
  return ContaGrado(fratello[x], k) + ContaGrado(figlio[x], k)
```

Interval Tree

Si definisca Interval Tree un albero binario di ricerca le cui entry sono intervalli $[a, b]$, con $a \leq b$, sulla retta reale. La chiave di una entry $[a, b]$, è rappresentata dall'estremo sinistro a dell'intervallo. Nell'albero non possono esistere due intervalli $[a, b]$ e $[c, d]$ in cui il primo sia interamente contenuto nel secondo, cioè con $c \leq a \leq b \leq d$.
(a) Disegnare l'Interval Tree costruito inserendo nell'ordine i seguenti intervalli, a partire dall'albero vuoto: $[9, 12]$, $[4, 7]$, $[17, 18]$, $[1, 3]$, $[8, 10]$, $[14, 16]$, $[20, 21]$, $[2, 4]$, $[13, 15]$.
(b)Descrivere un algoritmo efficiente che dato un intervallo $[a, b]$ e un Interval Tree T determina se $[a, b]$ può essere inserito in T, ovvero se non esiste in T un intervallo che contiene interamente $[a, b]$ o che è contenuto interamente in $[a, b]$. Qual'è la complessità dell'algoritmo?
Soluzione:



```
InsertCheck (real a, real b, node v)

if (v=NIL) then
  return YES
[c,d] ← v.element().getValue()
if ((c ≤ a ≤ b ≤ d)or(a ≤ c ≤ d ≤ b)) then
  return NO
if (a<c) then
  return InsertCheck(a,b,left(v))
else
  return InsertCheck(a,b,right(v))
```

ContaRip

Si scriva lo pseudocodice di un algoritmo che dato un BST T restituisce il numero massimo di ripetizioni di una chiave in T.

Soluzione:

```
ContaRip (T)

x ← root[T] if (x=NIL) then
  return 0
y ← min(x)
max_count ← 1
count ← 1
k ← key[y]
z ← succ[y]
while z! = NIL do
  if (k = key[z]) then
    count ← count + 1
  else
    count ← 1
    k ← key[z]
  z ← succ[z]
return max_count
```

Lezione 5 (07.04.2010)

Controlla Valore

Dato un albero binario di ricerca T, bilanciato e contenente chiavi tutte distinte, e due chiavi $k_1 < k_2$, scrivere un algoritmo che restituisce *true* se e solo se per ogni nodo di T se $key[x] = k_1$, allora non esiste alcun nodo y in T tale che $k_1 < key[y] < k_2$.
Dire qual'è la complessità dell'algoritmo rispetto al numero delle chiavi memorizzate nell'albero e spiegare perchè è corretto.
Soluzione:

```
ControllaValore (x,k1,k2)
---------------------------------
z ← BSTSeach(x, k1)
if (z = NIL) or (succ(z) = NIL) then
  return true
else
  return (key[succ(z)] ≥ k2)
```

Controlla Valore2

Dato un albero binario di ricerca T e due chiavi $k_1 < k_2$, scrivere un algoritmo che restituisce *true* se e solo se T contiene almeno una chiave k tale che $k_| \leq k \leq k_2$.
Soluzione:

```
ControllaValore2 (x,k1,k2)
---------------------------------
if x = NIL then
  return false
if key[x] = k1 then
  return true
if key[x] < k1 then
  return ControllaValore2(right[x], k1, k2)
if (key[x] > k1) and (key[x] ≤ k2) then
  return true
if (key[x] > k1) and (key[x] > k2) then
  return ControllaValore2(left[x], k1, k2)
```

Foglie con costo minimo

Dato un albero con radice T, con insieme dei nodi V, e una funzione peso $p : V \rightarrow R$, chiamiamo costo di un nodo v il valore

$$c(v) = \begin{cases} p(v) & \text{se } v \text{ è radice} \\ p(v) + c(padre(v)) & \end{cases}$$

Dato un albero con radice T e per ogni nodo v di T il peso $p(v)$, scrivere un algoritmo in pseudocodice che ritorni l'insieme delle foglie di T che hanno costo minimo.
Visita in pre-ordine per calcolare i costi e memorizzare il minimo.
Soluzione:

```
Main()
  m ← ∞
  F ← emptylist
  C ← p(r)
  Visita(r)
  S ← emptylist
  forall the (v, x) ∈ F do
  |   if x = m then
  |   └ S ← InserisciInTesta(S, v)
  └
  return S
```

```
Visita()
  if L(v) == empty then
  |   F ← InserisciInTesta(F, (v, C))
  |   if C < m then
  |   └ m ← C
  └
  else
  |   forall the w ∈ L(v) do
  |   |   C ← C + p(v)
  |   └   Visita(w)
  └
  C ← C − p(v)

  // V: insieme dei nodi di T
  // r: radice di T
  // L(v): lista dei figli di v
  // c(v): costo del nodo v
  // F: lista delle foglie di T con relativo costo
  // m: valore minimo dei costi nella lista F
  // C: costo del nodo corrente
```

Grafi orientati e non orientati: definizione

Un grafo orientato G è una coppia (V, E) dove:

- Insieme finito dei vertici V
- Insieme degli archi E: relazione binaria tra vertici

Un grafo non orientato G è una coppia (V, E) dove:

- Insieme finito dei vertici V
- Insieme degli archi E: coppie non ordinate

Rappresentazione grafi

Poniamo

- $n = |V|$ numero di nodi
- $m = |E|$ numero di archi

Matrice di adiacenza

- Spazio richiesto $O(n^2)$
- Verificare se il vertice u è adiacente a v richiede tempo $O(1)$
- Elencare tutti gli archi costa $O(n^2)$

$$m_{uv} = \begin{cases} 1 & \text{se } (u,v) \in E \\ 0 & \text{se } (u,v) \notin E \end{cases}$$

In-degree & out-degree

Data una rappresentazione con liste di adiacenza, quanto tempo/spazio occorre per calcolare e stampare il grado uscente e il grado entrante di ogni vertice?

```
InOut (Node *A[],int size)
in_count[] =int[n]; out_count[] =int[n];
for (int i = 0; i < n; i++) do
    Node* curr = A[i]; out_count = 0;
    while curr ≠ NIL do
        out_cout[i]++;
        curr = curr->next;
        in_count[curr->val]++;
print(out_count); print(in_count);
```

Universal sink

Data una rappresentazione con matrice di adiacenza, mostrare un algoritmo che opera in tempo $\Theta(V)$ in grado di determinare se un grafo orientato contiene un pozzo universale: ovvero un nodo con out-degree uguale a zero e in-degree uguale a $|V| - 1$.

1	*2*	*3*	*4*	*5*	*6*
0	1	0	1	0	0
0	0	1	0	0	0
0	0	0	0	1	0
1	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	0	0

```
UniversalSink(A[1..n])
  i ← 1
  candidate ← false
  while i < n and candidate = false do
    j ← i + 1
    while j ≤ n and A[i, j] = 0 do
      j ← j + 1
    if j>n then
      candidate ← true
    else
      j ← i
  rowtot = Σ_{j∈(1..n)−(i)} A[i, j]
  coltot = Σ_{j∈(1..n)−(i)} A[j, i]
  return rowtot = 0 and coltot = n − 1
```

Lezione 6 (15.04.2010)

Cycling

Descrivere un algoritmo che verifica se un grafo non orientato contiene o meno cicli. Tale algoritmo deve operare in tempo $O(V)$, indipendentemente da E.
Soluzione:

```
dfs (GRAPH G,NODE u, boolean[] visitato)
  visitato[u] ← true
  { esamina il nodo u (caso previsita) }
  foreach v ∈ G.adj(u) do
    { esamina l'arco (u, v) }
    if not visitato[v] then
      dfs(G, v, visitato)
    else
      return true
  { esamina il nodo u (caso postvisita) }
```

Distamax

Descrivere un algoritmo per il seguente problema. Dato in input un grafo orientato $G = (V, E)$ rappresentato tramite liste di adiacenza e un nodo s in V, restituire il numero dei nodi in V raggiungibili da s che si trovano alla massima distanza da s.
Soluzione:

```
MaxDist (V, E, s)
forall the v ∈ V do
  v.mark <- false
s.mak ← true
s.d = 0
queue ← s
current ← 0
count ← 0
while queue ≠ 0 do
  v ← queue.extract()
  forall the u ∈ v.adj do
    queue.add(u)
    if u.mark = false then
      u.mark ← true
      u.d ← v.d + 1
      if u.d > current then
        current ← u.d
        count ← 0
      count ← count + 1
```

Lezione 7 (21.04.2010)

Programmazione dinamica - Principi basilari

- Caratterizzare la struttura di una soluzione ottima
- Definire ricorsivamente il valore di una soluzione ottima
  - la soluzione ottima ad un problema contiene le soluzioni ottime ai sottoproblemi
- Calcolare il valore di una soluzione ottima "bottom-up" (cioè calcolando prima le soluzioni ai casi più semplici
  - Si usa una tabella per memorizzare le soluzioni dei sottoproblemi
  - Evitare di ripetere il lavoro più volte, utilizzando la tabella
- Costruire la (una) soluzione ottima

Longest Common Subsequence

$Z \in LCS(X, Y)$ è una sottosequenza di X e Y tale che, per ogni sottosequenza $W \in CS(X, Y)$, si ha che $|W| \leq |Z|$.
Date in input due sequenze di simboli X e Y, trovare la più lunga sottosequenza Z comune a X e Y.

$$X = (x_1, ..., x_m) \quad Y = (y_1, ..., y_n) \quad Z = (z_1, ..., > z_k) = LCS$$

Soluzione:

- $x_m = y_n \quad \rightarrow \quad z_k = x_m = y_n$ e $z(k-1) \in LCS(X(m-1), Y(n-1))$
- $x_m \neq y_n$ e $z_k \neq x_m \quad \rightarrow \quad Z \in LCS(X(m-1), Y)$
- $x_m \neq y_n$ e $z_k \neq y_n \quad \rightarrow \quad Z \in LCS(X, Y(n-1))$

	i	*0*	*1*	*2*	*3*	*4*	*5*	*6*
j			A	T	B	C	B	D
0		0	0	0	0	0	0	0
1	T	0	0	1	1	1	1	1
2	A	0	1	1	1	1	1	1
3	C	0	1	1	1	2	2	2
4	C	0	1	1		2	2	2
5	B	0	1	1				3
6	T	0	1	2			3	3

- se $x_m = y_n$, dobbiamo risolvere un sottoprovlema
  - $LCS\,(X(m-1), Y(n-1))$
  - la definizione ricorsiva è $LCS(X, Y) = LCS\,(X(m-1), Y(n-1))\,x_m$
- se $x_m \neq y_n$, dobbiamo risolvere 2 sottoproblemi
- ...

---

int LCS-Lenght (char *A, char *B)

---

if $*A ==' \backslash 0'$ **or** $*B ==' \backslash 0'$ **then**
  return 0
else if $*A == *B$ **then**
  return $1 + LCS - Lenght(A+1, B+1)$
else
  return $Max(LCS - Lenght(A+1, B), LCS - Lenght(A, B+1))$

Lezione 8 (28.04.2010)

Trovaddendi

Dati n interi positivi $x_1, x_2, ..., x_n$e due interi M e k vogliamo sapere se è possibile ottenere Mcome somma di al più k addendi scelti tra n numeri disponibili utilizzando eventualmente anche più volte gli stessi numeri.
Esempio. Dati i numeri $2, 5, 7, 8, 10$ e $k = 3$

- per $M = 26$ la risposta è SI $(8 + 8 + 10 = 26)$
- per $M = 12$ la risposta è SI $(5 + 7 = 12$ e $5 + 5 + 2 = 12$ e $2 + 10 = 12$ e $2 + 2 + 8 = 12)$
- per $M = 3$ la risposta è NO

Proporre un algoritmo che risolve il problema in tempo $O(Mn)$.
Soluzione:

- Si mantenga un vettore T di dimensione M tale che $T[i]$ =il numero minimo di addendi necessari oer ottenere i utilizzando numeri in $\{x_1, x_2, ..., x_n\}$, oppure $+\infty$ se non è possibile ottenere i dalla somma di numeri in $\{x_1, x_2, ..., x_n\}$.

$$T[i] = \begin{cases} +\infty & \text{se } i < min\{x_1, x_2, ..., x_n\} \\ 1 & \text{se } i \in \{x_1, x_2, ..., x_n\} \\ 1 + min_{1 \le j \le n, x_j < i} & \text{altrimenti} \end{cases}$$

```
Trovaddendi
for i ← 1 to M do
    T[i] ← +∞
    for j ← 1 to n do
        if x_j = i then
            T[i] ← 1
        if x_j < i and T[i] > T[i − x_j] + 1 then
            T[i] ← T[i − x_j] + 1
if T[M] ≤ k then
    return SI
else
    return NO
```

Somma di monete

Input: Un valore monetario V, ed un vettore di valori di monete $v[1..n]$, con$v[1] > v[2] > ... > v[n] = 1$
Output: Il minimo numero di monete il cui valore totale sia esattamente pari a V.
In altri termini, detto $a_i < 0$il numero di monete di valore $v[i]$che usiamo, vogliamo minimizzare il numero totale di monete usate, pari a:

$$a_1 + a_2 + ... + a_n \text{ sotto la condizione che } \sum_{i=1}^{n} a_i v[i] = V$$

Soluzione:
Sia $C(i, j)$ il minimo numero di monete necessario per esprimere la somma $j \le V$, usando monete di valore $v[i] > v[i+1] > ... > v[n]$. La soluzione sarà $C(1, V)$.
Consideriamo l'esempio con $V = 12$, e monete di valore $v[1] = 10$, $v[2] = 6$, $v[3] = 1$. L'indice di riga i specifica che sono disponibili le monete di valore $v[i], v[i+1], ..., v[n]$. L?indice di colonna j specifica il valore totale che si deve esprimere.

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
1	0	1	2	3	4	5	1	2	3	4	1	2	2
2	0	1	2	3	4	5	1	2	3	4	5	6	2
3	0	1	2	3	4	5	6	7	8	9	10	11	12

$$C(i, j) = \begin{cases} C(i+1, j) & \text{se } v[i] > j \\ min\{C(i+1, j), 1 + C(i, j - v[i])\} & \text{se } v[i] \le j \end{cases}$$

con i casi base: $C(n, j) = j, \ \forall j = 0, ..., V$

```
Sommamonete (v[1...n], V)

for j ← 0 to V do
    C(n, j) ← j
for i ← n − 1 downto 1 do
    for j ← 0 to V do
        if v[i] > j or C(i + 1, j) < 1 + C(i, j − v[i]) then
            C(i, j) ← C(i + 1, j)
        else
            C(i, j) ← C(i, j − v[i]) + 1
return C(1, V)
```

Lezione 9 (30.04.2010)

Business plan optimizer

Presso un'azienda sono disponibili per le prossime n settimane compiti che possono essere suddivisi in semplici o impegnativi. Il guadagno che si ricava dall'eseguire i compiti semplici della j-esima settimana è x_j, mentre quello che deriva dall'eseguire i compiti impegnativi è y_j.
Bisogna pianificare che tipo di compiti svolgere nelle prossime n settimane in modo da massimizzare il ricavo, tenendo conto che a ciascuna settimana in cui si è scelto di svolgere i compiti impegnativi deve seguire una settimana di riposo.
Ad esempio: per $X = 10\,1\,10\,10 \quad Y = 5\,50\,5\,1$

- la pianificazione $[x, x, x, x]$ che prevede di scegliere in ciascuna settimana i compiti semplici ha valore $10 + 1 + 10 + 10 = 31$
- la pianificazione $[y, *, y, *]$ che prevede di scegliere sempre i compiti impegnativi ha valore $5 + 5 = 10$
- la pianificazione che massimizza il ricavo è la $[x, y, *, x]$ che vale $10 + 50 + 10 = 70$ e prevede di scegliere i compiti impegnativi nella sola seconda settimana

Proporre un algoritmo che in tempo $O(n)$ calcola il valore della pianificazione a ricavo massimo (prm).
Soluzione:
Dobbiamo calcolare tre tipi di valori:

- $M[x, j]$: il valore della prm per le prime j settimane quando nella j-esima si sceglie di eseguire i compiti semplici
- $M[y, j]$: il valore della prm per le prime j settimane quando nella j-esima si sceglie di eseguire i compiti difficili
- $M[*, j]$: il valore della prm per le prime j settimane quando nella j-esima si è a riposo

Per $j > 2$ deve aversi:

- $M[x,j] = max\{M[x,j-1], M[*,j-1]\} + x_j$
- $M[y,j] = max\{M[x,j-1], M[*,j-1]\} + y_j$
- $M[*,j] = max\{M[x,j-1], M[y,j-1], M[*,j-1]\}$

Pertanto il valore della prm è dato da: $max\{M[x,n], M[y,n], M[*,n]\}$

Come passare un esame

Si consideri un esame con n problemi, ognuno dei quali vale $v[i]$ punti e richiede $t[i]$ minuti per essere risolto.
Trovare un algoritmo per risolvere il seguente problema: dati $v[1..n]$, $t[1..n]$e un valore V (sufficienza), trovare il sottoinsieme di problemi che richiede il minor tempo possibile e permette di passare l'esame.

1. Sia $T[i,v]$ il numero minimo di minuti richiesti per ottenere almeno v risolvendo un qualunque sottoinsieme dei primi i problemi. Scrivere un'espressione ricorsiva per $T[i,v]$ (compreso il caso base)
2. Scrivere l'algoritmo basato su programmazione dinamica.
3. Supponete ora che fare un esercizio in maniera parziale dia la possibilità di ottenere un voto parziale proporzionale. Descrivere un algoritmo di costo $O(n \log n)$ per risolvere il problema di pttenere l'insieme di problemi da risolvere per impiegare il tempo minimo per ottenere V.

Soluzione:

1. Ricorsione per $T[i,v]$
   - Per ottenere almeno 0 o un voto negativo, posso uscire subito: $T[i,v] = 0 : \forall i, \forall v \leq 0$
   - Se non faccio problemi, ma voglio ottenere v, ci vorrà un tempo infinito: $T[0,v] = +\infty : \forall v > 0$
   - Se considero il problema i, ci possono essere die casi: se lo risolvo, mi basta cercare di ottenere $v - v[i]$ con i primi $i-1$ problemi; altrimenti, devo ottenere $v$con i primi $i-1$ problemi: $T[i,v] = min\{t[i] + T[i-1, v - v[i]], T[i-1,v]\}$

2.

```
Studente-pigro (v[1...n], t[1...n], V)
for v ← 1 to V do
    T[0, v] ← +∞
for i ← 1 to n do
    for v ← 1 to V do
        if v − v[i] ≤ 0 then
            a ← t[i]
        else
            a ← t[i] + T[i − 1, v − v[i]]
        T[i, v] ← mina, T[i − 1, v]
return C(1, V)
```

3. Ordinare i problemi per $t[i]/v[i]$ crescente (o per $v[i]/t[i]$ decrescente) e iterare sulla lista fino a raggiungere v

Biblioteconomia dinamica

Un bibliotecario deve disporre n libri su opportuni scaffali. L'i-esimo libro, per $1 \leq i \leq n$, ha altezza h_i, e spessore s_i. Tutti gli scaffali hanno una stessa lunghezza L. I libri devono essere disposti secondo un ordine prefissato, in modo che la somma degli spessori dei libri su uno stesso scaffale non superiori a L. Gli scaffali possono essere posti a varie altezze. La distanza tra due scaffali consecutivi è pari all'latezza del libro più alto posto nello scaffale inferiore. Gli scaffali, inoltre, non devono necessariamente essere riempiti: il bibliotecario ha la possibilità di lasciare spazi vuoti al loro interno. Come può il bibliotecario partizionare i libri negli scaffali in modo da minimizzare l'altezza totale degli scaffali utilizzati?
Ad esempio, siano $n = 4$, $L = 4$, ed altezze e spessori come nella seguente tabella:

i	1	2	3	4
h_i	10	20	30	5
s_i	1	2	1	1

- Non è possibile assegnare tutti i libri ad uno stesso scaffale.
- La partizione che assegna ogni libro ad un diverso scaffale comporta un'altezza totale degli scaffali pari a $10 + 20 + 30 + 5 = 65$.
- La partizione che assegna il libro $\{1\}$al primo scaffale ed i libri $\{2, 3, 4\}$al secondo scaffale comporta un'altezza totale degli scaffali pari a $10 + max\{20, 30, 5\} = 40$.
- La partizione che minimizza l'altezza totale degli scaffali assegna i libri $\{1, 2, 3\}$al primo scaffale e il libro $\{4\}$al secondo scaffale e comporta un'altezza pari a $max\{10, 20, 30\} + 5 = 35$

Soluzione:
Studiamo il sottoproblema $P(i)$ che richiede di calcolare l'altezza minima degli scaffali necessari per disporre i primi i libri. Manteniamo le soluzioni di tali sottoproblemi in un array H di dimensione $n + 1$. Dunque $H[0] = 0$ e la nostra soluzione sarà $H[n]$. Per calcolare $H[i]$, per ogni j tale che $1 \leq j \leq i$, possiamo immaginare di disporre i libri da j a i in uno stesso scaffale (se possibile) ed i libri da 1 a $j - 1$ in maniera ottima in scaffali la cui altezza totale è data da $H[j - 1]$.

$$H[i] = min_{1 \leq j \leq i:\, s_j + \ldots + s_i \leq L}\{H[j-1] + max\{h_j, ..., h_i\}\}$$

```
Studente-pigro (n, L, hi, si)
H[0] ← 0
for i ← 1 to n do
    larghezza ← s_i
    altezza ← h_i
    min ← H[i − 1] + altezza
    j ← i − 1

    while j > 0 and larghezza + s_j ≤ L do
        larghezza ← larghezza + s_j
        altezza ← maxaltezza, h_j
        if min > H[j-1]+altezza then
            min ← H[j − 1] + altezza
        j ← j − 1
    H[i] ← min
return H[n]
```

Lezione 10 (05.05.2010)

Immobiliaristi golosi

Un'impresa edile termina la costruzione di n nuovi appartamenti e stipula un contratto con un'agenzia immobiliare, che si impegna ad acquistarli tutti. Gli appartamenti hanno diversi costi $c_1, ..., c_n$, ma il loro valore di mercato decresce con il passare del tempo: nel mese i l'appartamento x avrà un costo $c_x/2^{i-1}$. L'agenzia immobiliare si accorda sulle seguenti clausule:

- si impegna ad acquistare tutti gli appartamenti entro n mesi
- verrà acquistato almeno un appartamento al mese fino ad esaurimento degli appartamenti in vendita
- gli appartamenti acquistati in ogni mese potranno essere liberamente scelti tra quelli non ancora venduti

L'impresa edile, per ottenere almeno un certo guadagno minimo m dalla vendita di ogni appartamento, fa aggiungere al contratto una ulteriore clausola: se il prezzo di un appartamento diventa $< m$, l'impresa è svincolata dall'obbligo di cedere all'agenzia immobiliare quell'appartamento.
Assumendo che risulti $c_x > m$ per ogni appartamento x, l'agenzia immobiliare vuole garantirsi l'acquisto di tutti gli appartamenti minimizzando la spesa.
Ad esempio, siano $n = 4$, $m = 11$, $c_1 = 100$, $c_2 = 20$, $c_3 = 80$, $c_4 = 20$.

- L'acquisto di tutti gli immobili nel primo mese soddifsa il requisito dell'impresa edile (il guadagno su ogni appartamento è > 11) e comporta per l'agenzia una spesa $100+20+80+20 = 220$
- L'acquisto degli immobili 1 e 4 nel primo mese, dell'imobile 2 nel secondo mese e dell'immobile 3 nel terzo mese non garantisce la possibilità di acquistare tutti gli appartamenti: il guadagno

sull'immobile 2 sarebbe infatti $20/2 < 11$, e quindi l'agenzia non avrebbe la certezza di ottenere quell'immobile

- L'acquisto degli immobili 1, 2 e 4 nel primo mese e dell'immobile 3 nel terzo mese non è possibile, perchè nel secondo mese non viene acquistato alcun appartamento ma ci sono ancora appartamenti non venduti.
- L'acquisto degli immobili 1, 2 e 4 nel primo mese e dell'immobile 3 nel secondo mese è invece legale e comporta una spesa pari a $100 + 20 + 20 + 80/2 = 180$
- La spesa minima in questo esempio, ottenuta acquistando gli immobili 2 e 4 nel primo mese, l'immobile 3 nel secondo mese e l'immobile 1 nel terzo mese, è pari a $20+20+80/2+100/4 = 105$

Descrivere lo pseudocodice di un algoritmo che dia in output per ogni appartamento il mese in cui l'agenzia immobiliare può acquistarlo in modo da minimizzare la spesa e garantirsi comunque l'acquisto di tutti gli appartamenti. L'algoritmo deve avere un tempo di calcolo $O(n \log n)$.
Soluzione:

```
Immobiliaristi (n, m, c[])
i ← 1
mese ← 1
fattore ← 1
while i < n do
    sol[i] ← mese
    i ← i + 1
    while (c[i]/2 · fattore) < m do
        sol[i] ← mese
        i ← i + 1
    mese ← mese + 1
    fattore ← 2 · fattore
return sol
```

Binary substrings I

Scrivere in pseudo-codice una procedura che presi in input due interi n e k con $n \geq k \geq 2$ stampi tutte le stringhe binarie di lunghezza n le cui sottostringhe di simboli uguali hanno tutte linghezza diversa da k. Ad esempio per $n = 4$ e $k = 2$ la procedura deve stamparte (non necessariamente in quest'ordine): 0000, 0001, 0101, 0111, 1000, 1010, 1110, 1111.
La complessità della procedura deve essere $O(n \cdot D(n))$, dove $D(n)$ è il numero di stringhe da stampare.
Soluzione:

```
suf(n, k, sol[], i, suf)
if i=n then
    print(sol[1], ...)
else
    if i = 0 then
        sol[1] ← 0
        suf(n, k, sol, 1, 1)
        sol[1] ←
        suf(n, k, sol, 1, 1)
    else
        if (suf + 1 ≠ k) or (i + 1 ≠ n) then
            sol[i + 1] ← sol[i]
            suf(n, k, sol, i + 1, suf + 1)
        if suf ≠ k then
            sol[i + 1] ← (sol[i] + 1)mod2
            suf(n, k, sol, i + 1, 1)
```

Prima chiamata: $suf(n, k, sol, 0, 0)$

Binary substrings II

Scrivere in pseudo-codice una procedura che, presi in input due interi positivi n e k, con $k < n$, stampi tutte le sequenze binarie lunghe n che contengono almeno k simboli 1 consecutivi.
Ad esempio se $n = 4$ e $k = 2$ allora la procedura deve stampare (non necessariamente in quest'ordine): 0011, 0110, 1100, 1110, 1101, 1011, 0111, 1111.
La complessità della procedura deve essere $O(n \cdot D(n))$, dove $D(n)$ è il numero di sequenze da stampare.
Per decidere se inserire o meno un elemento nella soluzione parziale, senza compromettere la possibilità di ottenere una sequenza da stampare, si segue la seguente strategia:

- Se nella soluzione parziale sono presenti almeno k simboli 1 adiacenti, allora sia aggiungere un simbolo 0 cche aggiungere un simbolo 1 permetterà comunque di ottenere strighe da stampare
- Se nella soluzione parziale non sono presenti k simboli...

Soluzione:

```
cons(n, sol[], k, i, t, flag)
```

```
if i=n then
    print(sol[1], sol[2], ...sol[n])
else
    if flag = vero then
        sol[i+1] ← 0
        cons(n, sol, k, i+1, 0, vero)
        sol[i+1] ← 1
        cons(n, sol, k, i+1, t+1, vero)
    else
        if n − (i+1) ≥ k then
            sol[i+1] ← 0
            cons(n, sol, k, i+1, 0, falso)
        sol[i+1] ← 1
        if t = k − 1 then
            cons(n, sol, k, i+1, k, vero)
        else
            cons(n, sol, k, i+1, t+1, falso)
```

Prima chiamata: $cons(n, sol, k, 0, 0, falso)$

Lezione 11 (12.05.2010)

Ternary strings

Scrivere in pseudo-codice una procedura che, preso in input un intero positivo n, utilizzando le sole cifre $\{0, 1, 2\}$, stampi tutte le sequenze di lunghezza n la somma dei cui elementi sia n.
Ad esempio, se $n = 4$ la procedura deve stampare (non necessariamente in quest'ordine):
0022, 0112, 0121, 0202, 0211, 0220, 1012, 1021, 1102, 1111, 1120, 1201, 1210, 2002, 2011, 2020, 2101, 2110, 2200.
Soluzione:

```
ter(n, sol[], i, tot)
```

```
if i=n then
    print(sol[1], sol[2], ...sol[n])
else
    if tot + 2(n − i − 1) ≥ n then
        sol[i+1] ← 0
        ter(n, sol, i+1, tot)
    if (tot < n) and (tot + 2(n − i − 1) + 1 ≥ n) then
        sol[i+1] ← 1
        ter(n, sol, i+1, tot+1)
    if (tot < n − 1) and (tot + 2(n − i − 1) + 2 ≥ n) then
        sol[i+1] ← 2
        ter(n, sol, i+1, tot+2)
```

Prima chiamata: $ter(n, sol, 0, 0)$

Ricerca bidimensionale

Dato un intero k e una matrice M di $n \cdot n$ interi, i cui elementi di ogni riga e di ogni colonna sono ordinati in modo crescente, si vuole determinare se l'intero k compare tra gli elementi della matrice.

Il giustificatore

Un testo scritto è composto da n parole. Sia w, la lunghezza della i-esima parola (per semplicità trascuriamo gli spazi tra una parola e la successiva). Il nostro obbiettivo è di produrre un layout, ovvero una lista di parole distribuite su più righe, che sia ottimale. Naturalmente non si può modificare l'ordine delle parole. Definiamo la lunghezza di una riga come la somma delle lunghezze delle parole su quella riga. Detta L la massima lunghezza consentita, a ogni riga lunga K aoosiamo una penalità $= L - K$. Il nostro obbiettivo è dunque produrre un layout che minimizzi la penalità totale, ovvero la somma delle penalità delle varie righe.
Soluzione:

```
giustificatore
//W[1..n] contiene le n parole da inserire
k ← 0
currentLine = firstLine
for i ← 1 to n do
    if k + w_i ≤ L then
        currentLine.add(W[i])
        k ← k + w_i
    else
        currentLine ← currentLine.next()
        currentLine.add(W[i])
        k = w_i
```

Prima chiamata: $ter(n, sol, 0, 0)$

Project planner

Un progetto è caratterizzato dalla tripletta (i, t_i, c_i), dove i è l'indice del proggetto, mentre $t_i > 0$ e $c_i > 0$ sono il suo tempo e il suo costo. Un progetto è accettabile se non si dispone di un altro proggetto che abbia tempo e costo inferiori. Dato un insieme di n progetti, si vuole il sottoinsieme dei progetti accettabili.
Ad esempio, dati i progetti:$\{(1, 5, 100)\ (2, 7, 90)\ (3, 6, 80)\ (4, 9, 110)\}$ restiruire $\{(1, 5, 100)\ (3, 6, 80)\}$.
Descrivere in pseudo codice un algoritmo che risolve il problema.

Lezione 12 (19.05.2010)

Lezione 13 (26.05.2010)

K-path

Scrivere in pseudo-codice una procedura che, preso in input un grafo diretto $G = (\{1, ..., n\}, E)$, due suoi vertici $u <> v$, e un intero k, $1 \leq k \leq n$, stampa tutti e soli i cammini da ua v che passano per j nodi intermedi (distinti).
Ad esempio, se G è un grafo completo di 5 nodi, $u = 2$, $v = 3$ e $k = 2$, la procedura dovrà stampare: $(2, 1, 4, 3)$, $(2, 1, 5, 3)$, $(2, 4, 1, 3)$, $(2, 4, 5, 3)$, $(2, 5, 1, 3)$, $(2, 5, 4, 3)$.

Calcetto

Si supponga di dover organizzare una partita di calcetto e di dover dividere i partecipanti in due squadre il più possibile bilanciate. In particolare si consideri un insieme di $2n$ partecipanti e per ogni $i = 1, ..., 2n$ sia $b_i$un coefficiente intero che identifica la bravura del giocatore i-esimo. Vogliamo organizzare due squadre il più possibile equilibrate (rispetto alla somma delle bravure). Proggettare un algoritmo che dica se è possibile suddividere i giocatori in due squadre di n giocatori con la stessa bravura. Discutere la complessità e specificare la tecnica utilizzata.
Soluzione:

```
Calcetto (b[],C,m,s,h)
---------------------------------------------------------------
if s = 0 and m = 0 then
  return true
if h = 0 then
  return false
if C(m, s, h) = NIL then
  C(m, s, h) = Calcetto(b, C, m, s, h − 1) and Calcetto(b, C, m − b[h], s − 1, h − 1)
  return C(m, s, h)
```