# ESERCIZIO 1 APPELLO DEL 22/01/2009

- Soluzione banale, che non sfrutta l'ordinamento dell'array V: provare tutte le coppie di nodi

```
count ← 0
for i=1 to m
   for j= i+1 to m
      if (V[i]+V[j]> K)
      then count++
return count
```

Tempo di esecuzione: $\Theta(m^2)$!

- Cerchiamo di sfruttare l'ordinamento di V per progettare un algoritmo più efficiente: per ogni elemento V[i], individuiamo il più piccolo indice j tale che V[i]+ V[j] > K. Cerchiamo j in [i+1, m+1] ⊛, anziché in [1, m+1], in modo da evitare di contare una stessa coppia due volte. L'indice j può essere trovato con un approccio tipo ricerca binaria:

⊛ Se $j = m+1$, allora $\forall j$ risulta $V[i]+V[j] \leq K$

$O(\log m)$

```
a ← i+1
b ← m
while (a<b) do
   m ← (a+b)/2
   if (V[i]+ V[m] ≤ K) then a ← m
   else b ← m
if (V[i]+ V[a] ≤ K) then j ← a+1
                    else j ← a
```

L'intero algoritmo è quindi:

$O(n \log n)$

```
count = 0
for i = 1 to n
    trova j come mostrato sopra          // O(log n)
    count += n - j + 1
return count
```

- Si può progettare un algoritmo con tempo di esecuzione $\Theta(n)$, sfruttando una duplice scansione di V da sinistra verso destra e da destra verso sinistra (come nell'algoritmo partition del quicksort)

```
i ← 1
j ← n
while (i<j) do
    if (V[i] + V[j] > K)
    then j--
    else {
        count += n - j
        i++
    }
while (i<n) do
    count += n - i
    i++
```

Ad es., se V = 10 20 30 40 50 60

e K = 60, il primo while conta le tre coppie (10, 60), (20, 60) e (20, 50); il secondo while - che

parte con $i = 3$, conta tutte le coppie rimanenti.
Quando si esegue il secondo while siamo certi che, $\forall j > i$, $V[i] + V[j] > K$: non è quindi necessario effettuare la verifica. Nell'esempio, le coppie contate nel secondo while sono
(30, 40), (30, 50), (30, 60), (40, 50), (40, 60), (50, 60)

ESERCIZIO 2 APPELLO 22/01/2009

- proc1 è iterativa e contiene due cicli nidificati. Poichè Istr richiede tempo di esecuzione $O(1)$, il tempo di esecuzione di proc1 è
$$f(n) = \Theta(|T| \cdot |S|) = \Theta(n \log n)$$

- mainProc esegue tre chiamate ricorsive. La prima avviene su
$$\left(i + \frac{numElem}{4}\right) - i = \frac{numElem}{4} \text{ elementi}$$
La seconda avviene su
$$\left(i + \frac{numElem}{2}\right) - \left(i + \frac{numElem}{4}\right) = \frac{numElem}{4}$$
elementi. La terza avviene su:
$$f - \left(f - \frac{numElem}{4}\right) = \frac{numElem}{4} \text{ elementi.}$$
Il resto delle operazioni richiede tempo $\Theta(n \log n)$

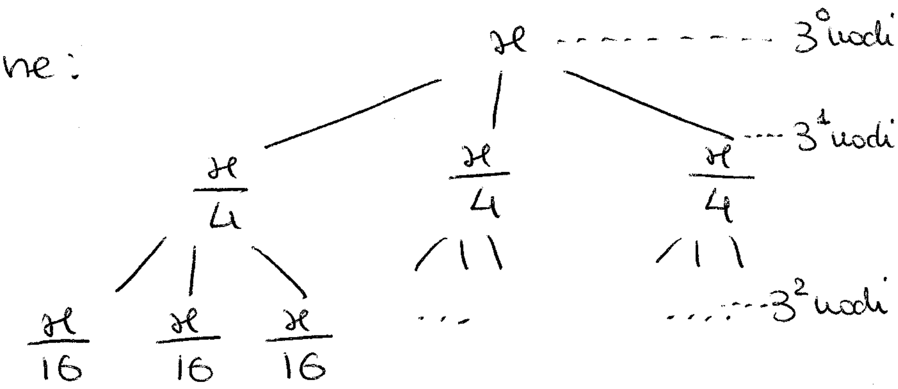
La ricorrenza è quindi:
$$T(x) = 3T\left(\frac{x}{4}\right) + \Theta(n \log n)$$

con $x = f - i + 1$.

Notare che, la chiamata proc1 richiede sempre tempo $\Theta(n \log n)$, indipendentemente da $i$ e $f$, e quindi indipendentemente dalla dimensione dell'input nella chiamata ricorsiva.

Per calcolare il tempo di esecuzione di mainProc basta quindi calcolare il # di nodi nell'albero della ricorsione, e moltiplicare tale numero per $\Theta(n \log n)$ che è il tempo speso su ciascun nodo.

Albero della ricorsione:




$$\#\text{nodi} = \sum_{i=0}^{\#\text{levels}} 3^i$$

Sia $k$ il massimo livello. I nodi sul livello $k$ hanno dimensione $\frac{n}{4^k} = 1 \Rightarrow k = \log_4 n$

Quindi:

$$\#\text{nodi} = \sum_{i=0}^{\log_4 n} 3^i = \frac{3^{\log_4 n + 1} - 1}{3 - 1} = \Theta\left(3^{\log_4 n}\right)$$

$$3^{\log_4 n} = 3^{\frac{\log_3 n}{\log_3 4}} = \left(3^{\log_3 n}\right)^{\frac{1}{\log_3 4}} = n^{\frac{1}{\log_3 4}} = n^{\log_4 3}$$

Usando le proprietà dei logaritmi

Concludendo, il tempo di esecuzione di mainProc su input di dimensione $n$ è:

$$T(n) = \Theta(n \log n) \cdot \Theta\left(n^{\log_4 3}\right) = \Theta\left(n^{1+\log_4 3} \log n\right)$$