Divide Et Impera

Sergio Flesca[1]

[1] DIMES - Università della Calabria, Via P. Bucci, 87036 Rende (CS) - Italy
flesca@dimes.unical.it

Outline

1. La tecnica divide et impera
   - Moltiplicazioni di interi con dimensione infinita
   - Moltiplicazione di matrici

Divide Et Impera

Gli algoritmi realizzati utilizzando questa tecnica seguono uno schema comune, per risolvere un'istanza di dimensione n fanno i seguenti passi:

1. Se l'istanza è di dimensione banale si risolve direttamente (oppure utilizzando algoritmi efficienti su problemi di piccole dimensioni)
2. Si suddivide l'istanza corrente in *"a"* istanze dello stesso problema che hanno dimensione più piccola rispetto all'istanza corrente (sotto-istanze), che hanno dimensione
   - $\frac{n}{c}$, oppure
   - $n - k$
3. Le sottoistanze vengono risolte applicando lo stesso algoritmo
4. Le soluzioni delle sottoistanze vengono combinate per ottenere la soluzione dell'istanza corrente

Divide Et Impera

Gli algoritmi realizzati utilizzando questa tecnica seguono uno schema comune, per risolvere un'istanza di dimensione n fanno i seguenti passi:

1. Se l'istanza è di dimensione banale si risolve direttamente (oppure utilizzando algoritmi efficienti su problemi di piccole dimensioni)
2. Si suddivide l'istanza corrente in *"a"* istanze dello stesso problema che hanno dimensione più piccola rispetto all'istanza corrente (sotto-istanze), che hanno dimensione
   - $\frac{n}{c}$, oppure
   - $n - k$
3. Le sottoistanze vengono risolte applicando lo stesso algoritmo
4. Le soluzioni delle sottoistanze vengono combinate per ottenere la soluzione dell'istanza corrente

Divide Et Impera

Gli algoritmi realizzati utilizzando questa tecnica seguono uno schema comune, per risolvere un'istanza di dimensione n fanno i seguenti passi:

1. Se l'istanza è di dimensione banale si risolve direttamente (oppure utilizzando algoritmi efficienti su problemi di piccole dimensioni)
2. Si suddivide l'istanza corrente in *"a"* istanze dello stesso problema che hanno dimensione più piccola rispetto all'istanza corrente (sotto-istanze), che hanno dimensione
   - $\frac{n}{c}$, oppure
   - $n - k$
3. Le sottoistanze vengono risolte applicando lo stesso algoritmo
4. Le soluzioni delle sottoistanze vengono combinate per ottenere la soluzione dell'istanza corrente

Divide Et Impera

Gli algoritmi realizzati utilizzando questa tecnica seguono uno schema comune, per risolvere un'istanza di dimensione n fanno i seguenti passi:

1. Se l'istanza è di dimensione banale si risolve direttamente (oppure utilizzando algoritmi efficienti su problemi di piccole dimensioni)
2. Si suddivide l'istanza corrente in *"a"* istanze dello stesso problema che hanno dimensione più piccola rispetto all'istanza corrente (sotto-istanze), che hanno dimensione
   - $\frac{n}{c}$, oppure
   - $n - k$
3. Le sottoistanze vengono risolte applicando lo stesso algoritmo
4. Le soluzioni delle sottoistanze vengono combinate per ottenere la soluzione dell'istanza corrente

Divide Et Impera

Gli algoritmi realizzati utilizzando questa tecnica seguono uno schema comune, per risolvere un'istanza di dimensione n fanno i seguenti passi:

1. Se l'istanza è di dimensione banale si risolve direttamente (oppure utilizzando algoritmi efficienti su problemi di piccole dimensioni)
2. Si suddivide l'istanza corrente in *"a"* istanze dello stesso problema che hanno dimensione più piccola rispetto all'istanza corrente (sotto-istanze), che hanno dimensione
   - $\frac{n}{c}$, oppure
   - $n - k$
3. Le sottoistanze vengono risolte applicando lo stesso algoritmo
4. Le soluzioni delle sottoistanze vengono combinate per ottenere la soluzione dell'istanza corrente

Divide Et Impera

Gli algoritmi realizzati utilizzando questa tecnica seguono uno schema comune, per risolvere un'istanza di dimensione n fanno i seguenti passi:

1. Se l'istanza è di dimensione banale si risolve direttamente (oppure utilizzando algoritmi efficienti su problemi di piccole dimensioni)
2. Si suddivide l'istanza corrente in *"a"* istanze dello stesso problema che hanno dimensione più piccola rispetto all'istanza corrente (sotto-istanze), che hanno dimensione
   - $\frac{n}{c}$, oppure
   - $n - k$
3. Le sottoistanze vengono risolte applicando lo stesso algoritmo
4. Le soluzioni delle sottoistanze vengono combinate per ottenere la soluzione dell'istanza corrente

Classi di algoritmi DI

- Gli algoritmi che al passo 2 della tecnica suddividono il problema in a sottoistanze di dimensione $\frac{n}{c}$ sono detti di tipo 1,
- Gli altri sono detti di tipo 2

Gli algoritmi di tipo 1 sono nella maggior parte dei casi più efficienti degli algoritmi di tipo 2.

Schema di algoritmo DI

```
public abstract class ProblemaDI {

public SoluzioneDI risolviDI() {
  if (dimensioneBanale())
    return risolviDirettamente();
  ProblemaDI[] sottoproblemi = dividi();
  SoluzioneDI[] sottosoluzioni = new SoluzioneDI[sottoproblemi.length];
  for (int i = 0; i<sottoproblemi.length; i++)
    sottosoluzioni[i] = sottoproblemi[i].risolviDI();
  return combina(sottosoluzioni);
}

protected abstract SoluzioneDI combina(SoluzioneDI[] sottosoluzioni);

protected abstract ProblemaDI[] dividi();

protected abstract boolean dimensioneBanale();

protected abstract SoluzioneDI risolviDirettamente();

}
```

Complessità algoritmi DI tipo 1

Supponiamo che l'algoritmo sia caratterizzato dalle seguenti costanti:

- a: numero di sottoistanze
- c: la costante che regola la dimensione delle sottoistanze $\frac{n}{c}$
- d: la complessita di dividi e combina è $\theta(n^d)$

La complessità dell'algoritmo (T) è descritta dalla seguente equazione di ricorrenza

$$\begin{array}{ll} T(n) = aT(\frac{n}{c}) + bn^d & \text{se } n > 1 \\ T(n) = b & \text{se } n \leq 1 \end{array}$$

Complessità algoritmi DI tipo 2

Supponiamo che l'algoritmo sia caratterizzato dalle seguenti costanti:

- a: numero di sottoistanze
- k: la costante che regola la dimensione delle sottoistanze $n-k$
- d: la complessita di dividi e combina è $\theta(n^d)$

La complessità dell'algoritmo (T) è descritta dalla seguente equazione di ricorrenza

$$\begin{array}{ll} T(n) = aT(n-k) + bn^d & \text{se } n > 1 \\ T(n) = b & \text{se } n \leq 1 \end{array}$$

Teorema delle ricorrenze (Complessita tipo 1)

Abbiamo un algoritmo caratterizzato da queste equazioni di riccorrenza

$$\begin{array}{lll} T(n) = aT\left(\frac{n}{c}\right) + bn^d & \text{se } n > 1 & (1) \\ T(n) = b & \text{se } n \leq 1 & (2) \end{array}$$

Teorema delle ricorrenze (Complessita tipo 1)

Abbiamo un algoritmo caratterizzato da queste equazioni di riccorrenza

$$
\begin{array}{lll}
T(n) = aT\left(\frac{n}{c}\right) + bn^d & \text{se } n > 1 & (1) \\
T(n) = b & \text{se } n \leq 1 & (2)
\end{array}
$$

utilizziamo la prima equazione per calcolare il termine $T(\frac{n}{c})$ che appare al suo interno, ottenendo $T\left(\frac{n}{c}\right) = aT\left(\frac{n}{c^2}\right) + b\left(\frac{n}{c}\right)^d$, e andiamo a sostituirlo nella prima equazione.

Teorema delle ricorrenze (Complessita tipo 1)

Abbiamo un algoritmo caratterizzato da queste equazioni di riccorrenza

$$\begin{array}{lll} T(n) = aT\left(\frac{n}{c}\right) + bn^d & \text{se } n > 1 & (1) \\ T(n) = b & \text{se } n \leq 1 & (2) \end{array}$$

utilizziamo la prima equazione per calcolare il termine $T(\frac{n}{c})$ che appare al suo interno, ottenendo $T\left(\frac{n}{c}\right) = aT\left(\frac{n}{c^2}\right) + b\left(\frac{n}{c}\right)^d$, e andiamo a sostituirlo nella prima equazione.

$$T(n) = aT\left(\frac{n}{c}\right) + bn^d = a\left(aT\left(\frac{n}{c^2}\right) + b\left(\frac{n}{c}\right)^d\right) + bn^d$$

ovverosia svolgendo il prodotto

Teorema delle ricorrenze (Complessita tipo 1)

Abbiamo un algoritmo caratterizzato da queste equazioni di riccorrenza

$$\begin{array}{lll} T(n) = aT\left(\frac{n}{c}\right) + bn^d & \text{se } n > 1 & (1) \\ T(n) = b & \text{se } n \leq 1 & (2) \end{array}$$

utilizziamo la prima equazione per calcolare il termine $T(\frac{n}{c})$ che appare al suo interno, ottenendo $T\left(\frac{n}{c}\right) = aT\left(\frac{n}{c^2}\right) + b\left(\frac{n}{c}\right)^d$, e andiamo a sostituirlo nella prima equazione.

$$T(n) = aT\left(\frac{n}{c}\right) + bn^d = a\left(aT\left(\frac{n}{c^2}\right) + b\left(\frac{n}{c}\right)^d\right) + bn^d$$

ovverosia svolgendo il prodotto

$$T(n) = a\left(aT\left(\frac{n}{c^2}\right) + b\left(\frac{n}{c}\right)^d\right) + bn^d = a^2T\left(\frac{n}{c^2}\right) + ab\left(\frac{n}{c}\right)^d + bn^d$$

Teorema delle ricorrenze (Complessita tipo 1)

$$T(n) = a^2 T\left(\frac{n}{c^2}\right) + ab\left(\frac{n}{c}\right)^d + bn^d = a^2 T\left(\frac{n}{c^2}\right) + ab\left(\frac{n}{c}\right)^d + a^0 b\left(\frac{n}{c^0}\right)^d =$$

Teorema delle ricorrenze (Complessita tipo 1)

$$T(n) = a^2 T\left(\frac{n}{c^2}\right) + ab\left(\frac{n}{c}\right)^d + bn^d = a^2 T\left(\frac{n}{c^2}\right) + ab\left(\frac{n}{c}\right)^d + a^0 b\left(\frac{n}{c^0}\right)^d =$$

$$= a^2 T\left(\frac{n}{c^2}\right) + \sum_{j=0}^{1} a^j b\left(\frac{n}{c^j}\right)^d$$

Teorema delle ricorrenze (Complessita tipo 1)

$$T(n) = a^2 T\left(\frac{n}{c^2}\right) + ab\left(\frac{n}{c}\right)^d + bn^d = a^2 T\left(\frac{n}{c^2}\right) + ab\left(\frac{n}{c}\right)^d + a^0 b\left(\frac{n}{c^0}\right)^d =$$

$$= a^2 T\left(\frac{n}{c^2}\right) + \sum_{j=0}^{1} a^j b\left(\frac{n}{c^j}\right)^d$$

ovverosia riassumendo

$$T(n) = a^2 T\left(\frac{n}{c^2}\right) + \sum_{j=0}^{1} a^j b\left(\frac{n}{c^j}\right)^d$$

ma al primo passo avevamo

$$T(n) = aT\left(\frac{n}{c}\right) + bn^d = a^1 T\left(\frac{n}{c^1}\right) + \sum_{j=0}^{0} a^j b\left(\frac{n}{c^j}\right)^d$$

Teorema delle ricorrenze (Complessita tipo 1)

Confrontando le due Equazioni riportate di seguito:

$$T(n) = a^2 T\left(\frac{n}{c^2}\right) + \sum_{j=0}^{1} a^j b \left(\frac{n}{c^j}\right)^d$$

$$T(n) = a^1 T\left(\frac{n}{c^1}\right) + \sum_{j=0}^{0} a^j b \left(\frac{n}{c^j}\right)^d$$

Ci accorgiamo che potremmo dimostrare che applicando k passi di sostituzione potremmo ottenere la formula

$$T(n) = a^k T\left(\frac{n}{c^k}\right) + \sum_{j=0}^{k-1} a^j b \left(\frac{n}{c^j}\right)^d$$

Difatti la formula è valida per $k = 1$ (l'equazione di ricorrenza iniziale) quindi ci basta dimostrare che se vale per k vale per $k + 1$ e avremo dimostrato che rimane valida fintantoche possiamo strovare il valore di $T\left(\frac{n}{c^k}\right)$ applicando la prima delle due equazioni di ricorrenza (per $\frac{n}{c^k} > 1$).

Teorema delle ricorrenze (Complessita tipo 1)

$$T(n) = a^k T\left(\frac{n}{c^k}\right) + \sum_{j=0}^{k-1} a^j b \left(\frac{n}{c^j}\right)^d$$

sostituendo $T\left(\frac{n}{c^k}\right)$ con la prima equazione di ricorrenza otteniamo

$$T(n) = a^k \left(aT\left(\frac{n}{c^{k+1}}\right) + b\left(\frac{n}{c^k}\right)^d \right) + \sum_{j=0}^{k-1} a^j b \left(\frac{n}{c^j}\right)^d =$$

$$= a^{k+1} T\left(\frac{n}{c^{k+1}}\right) + a^k b\left(\frac{n}{c^k}\right)^d + \sum_{j=0}^{k-1} a^j b \left(\frac{n}{c^j}\right)^d =$$

$$= a^{k+1} T\left(\frac{n}{c^{k+1}}\right) + \sum_{j=0}^{k} a^j b \left(\frac{n}{c^j}\right)^d \text{ C.V.D.}$$

Teorema delle ricorrenze (Complessita tipo 1)

Quindi la formula

$$T(n) = a^k T\left(\frac{n}{c^k}\right) + \sum_{j=0}^{k-1} a^j b \left(\frac{n}{c^j}\right)^d$$

è valida dopo k sostituzioni, ma all'aumentare di k si verifichera che $\frac{n}{c^k} \leq 1$ e quindi $T\left(\frac{n}{c^k}\right)$ dovrà essere ottenuto utilizzando l'equazione 2, ovverosia $T\left(\frac{n}{c^k}\right) = b$, ottenendo cosi (per il più piccolo valore di k tale che $\frac{n}{c^k} \leq 1$)

$$T(n) = a^k b + \sum_{j=0}^{k-1} a^j b \left(\frac{n}{c^j}\right)^d = a^k b \left(\frac{n}{c^k}\right)^d + \sum_{j=0}^{k-1} a^j b \left(\frac{n}{c^j}\right)^d = \sum_{j=0}^{k} a^j b \left(\frac{n}{c^j}\right)^d$$

Poichè il più piccolo valore di k tale che $\frac{n}{c^k} \leq 1$ è il più piccolo valore di k tale che $n \leq c^k$ allora tale valore è $k = \lceil \log_c n \rceil$ e sostituendolo nella formula precedente otteniamo

$$T(n) = \sum_{j=0}^{\lceil \log_c n \rceil} a^j b \left(\frac{n}{c^j}\right)^d = \sum_{j=0}^{\lceil \log_c n \rceil} bn^d a^j \frac{1}{(c^j)^d} = \sum_{j=0}^{\lceil \log_c n \rceil} bn^d \frac{a^j}{(c^d)^j} = \sum_{j=0}^{\lceil \log_c n \rceil} bn^d \left(\frac{a}{c^d}\right)^j$$

Teorema delle ricorrenze (Complessita tipo 1)

Possiamo studiare la formula

$$T(n) = bn^d \sum_{j=0}^{\lceil \log_c n \rceil} \left(\frac{a}{c^d} \right)^j \quad (3)$$

ragionando per casi, ed i particolare ragionando sul valore di $\frac{a}{c^d}$

Teorema delle ricorrenze (Complessita tipo 1)

Possiamo studiare la formula

$$T(n) = bn^d \sum_{j=0}^{\lceil \log_c n \rceil} \left(\frac{a}{c^d}\right)^j \quad (3)$$

ragionando per casi, ed i particolare ragionando sul valore di $\frac{a}{c^d}$

1. $\frac{a}{c^d} = 1$, in questo caso $\sum_{j=0}^{\lceil \log_c n \rceil} \left(\frac{a}{c^d}\right)^j = \sum_{j=0}^{\lceil \log_c n \rceil} 1^j = \lceil \log_c n \rceil + 1$ e quindi $T(n) = bn^d \lceil \log_c n \rceil + 1$, cioè $T(n) \in \theta(n^d \log_c n)$

Teorema delle ricorrenze (Complessita tipo 1)

Possiamo studiare la formula

$$T(n) = bn^d \sum_{j=0}^{\lceil \log_c n \rceil} \left(\frac{a}{c^d}\right)^j \quad (3)$$

ragionando per casi, ed i particolare ragionando sul valore di $\frac{a}{c^d}$

1. $\frac{a}{c^d} = 1$, in questo caso $\sum_{j=0}^{\lceil \log_c n \rceil} \left(\frac{a}{c^d}\right)^j = \sum_{j=0}^{\lceil \log_c n \rceil} 1^j = \lceil \log_c n \rceil + 1$ e quindi $T(n) = bn^d \lceil \log_c n \rceil + 1$, cioè $T(n) \in \theta(n^d \log_c n)$
2. $\frac{a}{c^d} < 1$, poiche la serie $\sum_{j=0}^{+\infty} \alpha^j$ per $\alpha < 1$ è convergente allora esiste un valore costante C tale che $\sum_{j=0}^{+\infty} \left(\frac{a}{c^d}\right)^j \leq C$. Sostituendo alla sommatoria C nella forumla otteniamo $T(n) = bn^d C$, ovverosia $T(n) \in \theta(n^d)$

Teorema delle ricorrenze (Complessita tipo 1)

Possiamo studiare la formula

$$T(n) = bn^d \sum_{j=0}^{\lceil \log_c n \rceil} \left(\frac{a}{c^d}\right)^j \quad (3)$$

ragionando per casi, ed i particolare ragionando sul valore di $\frac{a}{c^d}$

1. $\frac{a}{c^d} = 1$, in questo caso $\sum_{j=0}^{\lceil \log_c n \rceil} \left(\frac{a}{c^d}\right)^j = \sum_{j=0}^{\lceil \log_c n \rceil} 1^j = \lceil \log_c n \rceil + 1$ e quindi $T(n) = bn^d \lceil \log_c n \rceil + 1$, cioè $T(n) \in \theta(n^d \log_c n)$
2. $\frac{a}{c^d} < 1$, poiche la serie $\sum_{j=0}^{+\infty} \alpha^j$ per $\alpha < 1$ è convergente allora esiste un valore costante C tale che $\sum_{j=0}^{+\infty} \left(\frac{a}{c^d}\right)^j \leq C$. Sostituendo alla sommatoria C nella forumla otteniamo $T(n) = bn^d C$, ovverosia $T(n) \in \theta(n^d)$
3. $\frac{a}{c^d} > 1$, per quest'ultimo caso dobbiamo lavorare un po di più!

Teorema delle ricorrenze (caso $\frac{a}{c^d} > 1$)

Possiamo dimostrare che per $\alpha > 1$ vale che $\sum_{i=0}^{k} \alpha^i = \frac{\alpha^{k+1}-1}{\alpha-1}$, infatti se moltiplichiamo $\sum_{i=0}^{k} \alpha^i$ per α otteniamo $\alpha \sum_{i=0}^{k} \alpha^i = \sum_{i=1}^{k+1} \alpha^i$, e se da quest'ultima formula sottraiamo $\sum_{i=0}^{k} \alpha^i$ ottenimao

$$(\alpha - 1) \sum_{i=0}^{k} \alpha^i = \alpha \sum_{i=0}^{k} \alpha^i - \sum_{i=0}^{k} \alpha^i = \alpha^{k+1} - 1$$

ovverosia

$$\sum_{i=0}^{k} \alpha^i = \frac{\alpha^{k+1} - 1}{\alpha - 1}$$

ed applicandolo nella formula (3)

$$T(n) = bn^d \sum_{j=0}^{\lceil \log_c n \rceil} \left(\frac{a}{c^d} \right)^j = bn^d \frac{\left(\frac{a}{c^d} \right)^{\lceil \log_c n \rceil + 1} - 1}{\frac{a}{c^d} - 1}$$

e quindi risulta $T(n) \in \theta \left(bn^d \left(\frac{a}{c^d} \right)^{\log_c n} \right)$

Teorema delle ricorrenze (caso $\frac{a}{c^d} > 1$)

Possiamo semplificare ulteriormente

$$bn^d \left(\frac{a}{c^d}\right)^{\log_c n} = bn^d \frac{a^{\log_c n}}{(c^d)^{\log_c n}} = bn^d \frac{a^{\log_c n}}{(c^{\log_c n})^d} = bn^d \frac{a^{\log_c n}}{n^d} = ba^{\log_c n}$$

Teorema delle ricorrenze (caso $\frac{a}{c^d} > 1$)

Possiamo semplificare ulteriormente

$$bn^d \left(\frac{a}{c^d}\right)^{\log_c n} = bn^d \frac{a^{\log_c n}}{(c^d)^{\log_c n}} = bn^d \frac{a^{\log_c n}}{(c^{\log_c n})^d} = bn^d \frac{a^{\log_c n}}{n^d} = ba^{\log_c n}$$

applicando la formula di cambio di base dei logaritmi $\log_c n = \log_c a \times \log_a n$ otteniamo

$$ba^{\log_c n} = ba^{\log_c a \times \log_a n} = bn^{\log_c a}$$

ovverosia $T(n) \in \theta(n^{\log_c a})$

Teorema delle ricorrenze

Dato un algoritmo DI di tipo 1 caratterizzato dalle seguenti costanti:

- a: numero di sottoistanze
- c: la costante che regola la dimensione delle sottoistanze $\frac{n}{c}$
- d: la complessita di dividi e combina è $\theta(n^d)$

La complessità dell'algoritmo è

1. $\theta(n^d \log_c n)$ se $\frac{a}{c^d} = 1$
2. $\theta(n^d)$ se $\frac{a}{c^d} < 1$
3. $\theta(n^{\log_c a})$ se $\frac{a}{c^d} > 1$

Esempi di algoritmi di tipo 1

- Ricerca Binaria
- Merge Sort
- Moltiplicazioni di interi con dimensione infinita
- Moltiplicazione di matrici

Ricerca Binaria

```
public class RicercaBinaria {

  /**
  * Riceve un vettore ordinato di interi v e un intero x e restituisce
  * la posizione di x in v o -1 se non presente
  */
  public static int ricercaBinariaRic(int[] v, int x) {
    return ricercaBinariaRic(v,x,0,v.length-1);
  }

  private static int ricercaBinariaRic(int[] v, int x, int inf, int sup) {
    if(sup>inf) return -1;
    int med = (inf+sup)/2;
    if (v[med]==x) return med;
    if (v[med]>x)
      return ricercaBinariaRic(v, x,inf,med-1);
    else
      return ricercaBinariaRic(v, x,med+1,sup);
  }

}
```

Ricerca Binaria

- $a = 1$, poichè cerchiamo l'elemento solo su una delle due metà del vettore
- $c = 2$, poichè dividiam il vettore in due metà
- $d = 0$ poiche per suddividere il vettore in due meta e comprendre qual'è la meta su cui spostarci utilizziamo $\theta(1)$ operazioni

$$\frac{a}{c^d} = \frac{1}{2^0} = 1$$

quindi siamo nel caso (1) del teorema e quindi la complessità è

$$\theta(n^d \log_c n) \Rightarrow \theta(n^0 \log_2 n) \Rightarrow \theta(\log_2 n)$$

Merge Sort

```
public class Ordinamento {

...

public static void mergeSort(int[] v){
  if (v!= null)
    mergeSort(v,0,v.length-1);
}

private static void mergeSort(int[] v, int in, int fin) {
  if(fin<=in)
    return;
  int med = (in+fin)/2;
  mergeSort(v, in, med);
  mergeSort(v, med+1,fin);
  merge(v,in,med,fin);
}

...
```

Merge Sort

```
...

private static void merge(int[] v, int in, int med, int fin) {
  int[] stage = new int[fin-in+1];
  int i=in,j=med+1, st=0;
  while((i<=med)&(j<=fin)){
    if(v[i]<v[j])
      stage[st++]=v[i++];
    else
      stage[st++]=v[j++];
  }
  for(;i<=med;i++)
    stage[st++]=v[i];
  for(;j<=fin;j++)
    stage[st++]=v[j];
  for(int k=0; k<stage.length;k++)
    v[in+k]=stage[k];
}
...
```

Merge Sort

- $a = 2$, poichè ordiniamo separatamente conlo stesso algoritmo le due metà del vettore
- $c = 2$, poichè dividiam il vettore in due metà
- $d = 1$ poiche la funzione merge ha complessità $\theta(n)$

$$\frac{a}{c^d} = \frac{2}{2^1} = 1$$

quindi siamo nel caso (1) del teorema e quindi la complessità è

$$\theta(n^d \log_c n) \Rightarrow \theta(n^1 \log_2 n) \Rightarrow \theta(n \log_2 n)$$

Merge Sort

- $a = 2$, poichè ordiniamo separatamente conlo stesso algoritmo le due metà del vettore
- $c = 2$, poichè dividiam il vettore in due metà
- $d = 1$ poiche la funzione merge ha complessità $\theta(n)$

$$\frac{a}{c^d} = \frac{2}{2^1} = 1$$

quindi siamo nel caso (1) del teorema e quindi la complessità è

$$\theta(n^d \log_c n) \Rightarrow \theta(n^1 \log_2 n) \Rightarrow \theta(n \log_2 n)$$

Qual'è la complessità spaziale?

Merge Sort

- $a = 2$, poichè ordiniamo separatamente conlo stesso algoritmo le due metà del vettore
- $c = 2$, poichè dividiam il vettore in due metà
- $d = 1$ poiche la funzione merge ha complessità $\theta(n)$

$$\frac{a}{c^d} = \frac{2}{2^1} = 1$$

quindi siamo nel caso (1) del teorema e quindi la complessità è

$$\theta(n^d \log_c n) \Rightarrow \theta(n^1 \log_2 n) \Rightarrow \theta(n \log_2 n)$$

Qual'è la complessità spaziale? $\theta(n + \log n)$ perchè dobbiamo allocare il vettore stage!

QuickSort come possibile miglioramento di MergeSort

Potremmo complicare un po la fase di suddivisione e spostare tutti gli elementi che nell'ordinamento si trovano nella meta sinistra in quella metà e gli altri nell'altra metà?

QuickSort come possibile miglioramento di MergeSort

Potremmo complicare un po la fase di suddivisione e spostare tutti gli elementi che nell'ordinamento si trovano nella meta sinistra in quella metà e gli altri nell'altra metà?

- Dobbiamo farlo al più in tempo lineare (altrimenti cambiamo le caratteristiche dell'algoritmo e quindi la complessità)

QuickSort come possibile miglioramento di MergeSort

Potremmo complicare un po la fase di suddivisione e spostare tutti gli elementi che nell'ordinamento si trovano nella meta sinistra in quella metà e gli altri nell'altra metà?

- Dobbiamo farlo al più in tempo lineare (altrimenti cambiamo le caratteristiche dell'algoritmo e quindi la complessità)
- Se conoscessimo il valore dell'elemento mediano potremmo farlo!

QuickSort come possibile miglioramento di MergeSort

Potremmo complicare un po la fase di suddivisione e spostare tutti gli elementi che nell'ordinamento si trovano nella meta sinistra in quella metà e gli altri nell'altra metà?

- Dobbiamo farlo al più in tempo lineare (altrimenti cambiamo le caratteristiche dell'algoritmo e quindi la complessità)
- Se conoscessimo il valore dell'elemento mediano potremmo farlo!
- Sfortunatamente calcolare il mediano è difficile quanto calcolare l'ordinamento! Che fare?

QuickSort come possibile miglioramento di MergeSort

Potremmo complicare un po la fase di suddivisione e spostare tutti gli elementi che nell'ordinamento si trovano nella meta sinistra in quella metà e gli altri nell'altra metà?

- Dobbiamo farlo al più in tempo lineare (altrimenti cambiamo le caratteristiche dell'algoritmo e quindi la complessità)
- Se conoscessimo il valore dell'elemento mediano potremmo farlo!
- Sfortunatamente calcolare il mediano è difficile quanto calcolare l'ordinamento! Che fare?
- Potremmo riuscire a "stimarlo", come?

QuickSort come possibile miglioramento di MergeSort

Potremmo complicare un po la fase di suddivisione e spostare tutti gli elementi che nell'ordinamento si trovano nella meta sinistra in quella metà e gli altri nell'altra metà?

- Dobbiamo farlo al più in tempo lineare (altrimenti cambiamo le caratteristiche dell'algoritmo e quindi la complessità)
- Se conoscessimo il valore dell'elemento mediano potremmo farlo!
- Sfortunatamente calcolare il mediano è difficile quanto calcolare l'ordinamento! Che fare?
- Potremmo riuscire a "stimarlo", come?
- Soluzione: scegliamo un elemento a caso nel vettore!

Quick Sort

```
...
private static void quickSort(int[] v, int in, int fin) {
  if(fin<=in)
    return;
  int p= partiziona(v,in,fin);
  quickSort(v,in,p-1);
  quickSort(v,p+1,fin);
}

private static int partiziona(int[] v, int in, int fin) {
  int rnd = (int) Math.floor(Math.random()*(fin-in+1));
  int tmp = v[in]; v[in]= v[rnd]; v[rnd]= tmp;
  int p=in;
  int inf=in+1, sup=fin;
  while(inf<sup){
    for(;(inf<=fin)&&(v[inf]<v[p]); inf++);
    for(;(sup>=in)&&(v[p]<=v[sup]);sup--);
    if(inf<sup){
      int t = v[inf]; v[inf]= v[sup]; v[sup]= t;
    }
  }
  int t = v[p]; v[p] = v[inf-1]; v[inf-1] = t;
  return inf-1;
}
...
```

QuickSort complessità caso peggiore

Nel caso peggiore siamo sfortunati e partiziona sceglie come pivot l'elemento più piccolo

QuickSort complessità caso peggiore

Nel caso peggiore siamo sfortunati e partiziona sceglie come pivot l'elemento più piccolo

$$T(n) = T(n-1) + bn \quad \text{se } n > 1 \quad (1)$$
$$T(n) = b \quad \text{se } n \leq 1 \quad (2)$$

QuickSort complessità caso peggiore

Nel caso peggiore siamo sfortunati e partiziona sceglie come pivot l'elemento più piccolo

$$
\begin{array}{lll}
T(n) = T(n-1) + bn & \text{se } n > 1 & (1) \\
T(n) = b & \text{se } n \leq 1 & (2)
\end{array}
$$

Non si applica il teorema delle ricorrenze!

Utilizziamo il metodo della sostituzione come abbiamo fatto nella prova del teorema.

QuickSort complessità caso peggiore

Nel caso peggiore siamo sfortunati e partiziona sceglie come pivot l'elemento più piccolo

$$T(n) = T(n-1) + bn \quad \text{se } n > 1 \quad (1)$$
$$T(n) = b \quad \text{se } n \leq 1 \quad (2)$$

Non si applica il teorema delle ricorrenze!

Utilizziamo il metodo della sostituzione come abbiamo fatto nella prova del teorema.

$$T(n) = T(n-1) + bn = T(n-2) + b(n-1) + bn = T(n-2) + b\sum_{j=0}^{1}(n-j) =$$
$$= T(n-3) + b(n-2) + b\sum_{j=0}^{1}(n-j) = T(n-3) + b\sum_{j=0}^{2}(n-j) = \cdots$$
$$\cdots = T(n-k) + b\sum_{j=0}^{k-1}(n-j)$$

QuickSort complessità caso peggiore

Nel caso peggiore siamo sfortunati e partiziona sceglie come pivot l'elemento più piccolo

$$
\begin{array}{lll}
T(n) = T(n-1) + bn & \text{se } n > 1 & (1) \\
T(n) = b & \text{se } n \leq 1 & (2)
\end{array}
$$

Non si applica il teorema delle ricorrenze!

Utilizziamo il metodo della sostituzione come abbiamo fatto nella prova del teorema.

$$
\begin{aligned}
&T(n) = T(n-1) + bn = T(n-2) + b(n-1) + bn = T(n-2) + b\sum_{j=0}^{1}(n-j) = \\
&= T(n-3) + b(n-2) + b\sum_{j=0}^{1}(n-j) = T(n-3) + b\sum_{j=0}^{2}(n-j) = \cdots \\
&\cdots = T(n-k) + b\sum_{j=0}^{k-1}(n-j)
\end{aligned}
$$

Quando $n-k=1 (k=n-1)$ si applica la seconda equazione quindi otteniamo:

$$
T(n) = T(1) + b\sum_{j=0}^{n-2}(n-j) = b(n-(n-1)) + b\sum_{j=0}^{n-2}(n-j) = b\sum_{j=0}^{n-1}(n-j) =
$$

$$
= b\sum_{j=1}^{n} j = b\frac{n(n-1)}{2} \in \theta(n^2)
$$

QuickSort Analisi del caso medio

Supponendo che l'algoritmo abbia la stessa probabilita di scegliere come pivot uno qualunque degli elementi (giustificata dalla scelta random) abbiamo la seguente equazione di ricorrenza:

$$T(n) = \sum_{p=0}^{n-1} \frac{1}{n}(n + T(p) + T(n-p-1))$$

ed osservando che nella sommatoria complessiva i termini $T(p)$ e $T(n-p-1)$ danno lo stesso contributo otteniamo

$$T(n) = n + \sum_{p=0}^{n-1} \frac{2}{n} T(p)$$

Possiamo dimostrare le $T(n) \leq 2n \log n$

QuickSort Analisi del caso medio

- Apllichiamo il metodo della sostituzione, ipotizzando che esista una costante α tale che $T(n) \leq \alpha n \log n$ e dimostriamolo applicando il principio di induzione generalizzato (ipotizando che nel caso $n = 1$ non sia necessario fare operazioni per risolvere il problema $T(1) = 0$).

QuickSort Analisi del caso medio

- Apllichiamo il metodo della sostituzione, ipotizzando che esista una costante α tale che $T(n) \leq \alpha n \log n$ e dimostriamolo applicando il principio di induzione generalizzato (ipotizzando che nel caso $n = 1$ non sia necessario fare operazioni per risolvere il problema $T(1) = 0$).
- Per n = 1, $T(1) = 0 = \alpha 1 \log 1$ per ogni α

QuickSort Analisi del caso medio

- Apllichiamo il metodo della sostituzione, ipotizzando che esista una costante α tale che $T(n) \leq \alpha n \log n$ e dimostriamolo applicando il principio di induzione generalizzato (ipotizando che nel caso $n = 1$ non sia necessario fare operazioni per risolvere il problema $T(1) = 0$).
- Per n = 1, $T(1) = 0 = \alpha 1 \log 1$ per ogni α
- Ipotizziamo quindi che per tutti gli $p < n$ vale che $T(p) \leq \alpha p \log p$ sostituiamo questo valore nell'equazione per $T(n)$, ottenendo

$$T(n) = n + \sum_{p=0}^{n-1} \frac{2}{n} \alpha p \log p = n + \frac{2\alpha}{n} \sum_{p=2}^{n-1} p \log p \leq n + \frac{2\alpha}{n} \int_{p=2}^{n-1} p \log p$$

- Applicando il metodo di integrazione per parti si ottine

$$T(n) \leq n + \frac{2\alpha}{n} \left(\frac{n^2 \log n}{2} - \frac{n^2}{4} - 2 \log 2 + 1 \right) \approx n + \alpha n \log n - \alpha \frac{n}{2} \leq \alpha n \log n$$

Quando $\alpha \geq 2$.

Eliminazione della ricorsione di coda

La tecnica dell'eliminazione di coda permette di sostituire l'ultima chiamata ricorsiva fatta (nel caso sia l'ultima cosa che facciamo nel metodo) con un while. Questo permette di eliminare la necessita di usare spazio di memoria per quella chiamata ricorsiva.

```
private static tipo metodo (parametri) {
  if(parametri di dimensione banale)
    return casobase(parametri);
  corpo(parametri)
  return metodo(parametrimodificati);
}
```

Possiamo cambiare questa funzione cosi:

```
private static tipo metodo (parametri) {
  while (parametri di dimensione non banale){
    corpo(parametri)
    parametri = parametrimodificati
  }
  return casobase(parametri);
}
```

Eliminazione della ricorsione di coda

La tecnica dell'eliminazione di coda permette di sostituire l'ultima chiamata ricorsiva fatta (nel caso sia l'ultima cosa che facciamo nel metodo) con un while. Questo permette di eliminare la necessita di usare spazio di memoria per quella chiamata ricorsiva.

```
private static tipo metodo (parametri) {
  if(parametri di dimensione banale)
    return casobase(parametri);
  corpo(parametri)
  return metodo(parametrimodificati);
}
```

Possiamo cambiare questa funzione cosi:

```
private static tipo metodo (parametri) {
  while (parametri di dimensione non banale){
    corpo(parametri)
    parametri = parametrimodificati
  }
  return casobase(parametri);
}
```

Eliminzione della ricorsione di coda del QuickSort

Applichiamo la tecnica al QuickSort

```
private static void quickSort(int[] v, int in, int fin) {
   if(fin<=in)
     return;
   int p= partiziona(v,in,fin);
   quickSort(v,in,p-1);
   quickSort(v,p+1,fin);
}
```

Applichiamo lo schema visto nella slide precedente e otteniamo

```
private static void quickSort(int[] v, int in, int fin) {
   while(!(fin<=in)){
     int p= partiziona(v,in,fin);
     quickSort(v,in,p-1);
     in=p+1; fin=fin;
   }
   return;
}
```

Abbiamo ridotto la Complessità Spaziale?

Eliminzione della ricorsione di coda del QuickSort

Applichiamo la tecnica al QuickSort

```
private static void quickSort(int[] v, int in, int fin) {
  if(fin<=in)
    return;
  int p= partiziona(v,in,fin);
  quickSort(v,in,p-1);
  quickSort(v,p+1,fin);
}
```

Applichiamo lo schema visto nella slide precedente e otteniamo

```
private static void quickSort(int[] v, int in, int fin) {
  while(!(fin<=in)){
    int p= partiziona(v,in,fin);
    quickSort(v,in,p-1);
    in=p+1; fin=fin;
  }
  return;
}
```

Abbiamo ridotto la Complessità Spaziale?Purtroppo no!

Eliminzione della ricorsione di coda del QuickSort

Dobbiamo fare una cosa più intelligente cerchiamo di eliminare la chiamata ricorsiva fatta sulla porzione più grande invece che l'ultima.

```
private static void quickSort(int[] v, int in, int fin) {
while(!(fin<=in)){
int p= partiziona(v,in,fin);
   if(p<((in+fin)/2)){
      quickSort(v,in,p-1);
      in=p+1; fin=fin;
   } else {
      quickSort(v,p+1,fin);
      fin=p-1; in=in;
   }
}
return;
}
```

Abbiamo ridotto la Complessità Spaziale?

Eliminzione della ricorsione di coda del QuickSort

Dobbiamo fare una cosa più intelligente cerchiamo di eliminare la chiamata ricorsiva fatta sulla porzione più grande invece che l'ultima.

```
private static void quickSort(int[] v, int in, int fin) {
while(!(fin<=in)){
int p= partiziona(v,in,fin);
  if(p<((in+fin)/2)){
    quickSort(v,in,p-1);
    in=p+1; fin=fin;
  } else {
    quickSort(v,p+1,fin);
    fin=p-1; in=in;
  }
}
return;
}
```

Abbiamo ridotto la Complessità Spaziale?Eccome $(\log n)$!

Moltiplicazioni di interi con dimensione infinita

L'algoritmo che abbiamo studiato alla scuola primaria per le moltiplicazioni di interi fa i seguenti passi: Per effettuare la moltiplicazione $x \times y$, dove x e y hanno n cifre, ed indicando con x_i (risp. y_i) la i-esima cifra di x la decompone nella seguente formula:

$$\sum_{i=0}^{n-1} x \times y_i \times B^i$$

dove B è la base del sistema di numerazione (10 per il sistema decimale, 2 per il binario)

- Ogni moltiplicazione $x \times y_i \times B^i$ ha complessità temporale $\theta(n)$
- quindi la complessità temporale dell'algoritmo è $\theta(n^2)$

Possiamo fare di meglio con la tecnica divide et impera?

Moltiplicazioni di interi con dimensione infinita

L'algoritmo che abbiamo studiato alla scuola primaria per le moltiplicazioni di interi fa i seguenti passi: Per effettuare la moltiplicazione $x \times y$, dove x e y hanno n cifre, ed indicando con x_i (risp. y_i) la i-esima cifra di x la decompone nella seguente formula:

$$\sum_{i=0}^{n-1} x \times y_i \times B^i$$

dove B è la base del sistema di numerazione (10 per il sistema decimale, 2 per il binario)

- Ogni moltiplicazione $x \times y_i \times B^i$ ha complessità temporale $\theta(n)$
- quindi la complessità temporale dell'algoritmo è $\theta(n^2)$

Possiamo fare di meglio con la tecnica divide et impera?

Moltiplicazioni di interi con dimensione infinita

L'algoritmo che abbiamo studiato alla scuola primaria per le moltiplicazioni di interi fa i seguenti passi: Per effettuare la moltiplicazione $x \times y$, dove x e y hanno n cifre, ed indicando con x_i (risp. y_i) la i-esima cifra di x la decompone nella seguente formula:

$$\sum_{i=0}^{n-1} x \times y_i \times B^i$$

dove B è la base del sistema di numerazione (10 per il sistema decimale, 2 per il binario)

- Ogni moltiplicazione $x \times y_i \times B^i$ ha complessità temporale $\theta(n)$
- quindi la complessità temporale dell'algoritmo è $\theta(n^2)$

Possiamo fare di meglio con la tecnica divide et impera?

Moltiplicazioni di interi con dimensione infinita

L'algoritmo che abbiamo studiato alla scuola primaria per le moltiplicazioni di interi fa i seguenti passi: Per effettuare la moltiplicazione $x \times y$, dove x e y hanno n cifre, ed indicando con x_i (risp. y_i) la i-esima cifra di x la decompone nella seguente formula:

$$\sum_{i=0}^{n-1} x \times y_i \times B^i$$

dove B è la base del sistema di numerazione (10 per il sistema decimale, 2 per il binario)

- Ogni moltiplicazione $x \times y_i \times B^i$ ha complessità temporale $\theta(n)$
- quindi la complessità temporale dell'algoritmo è $\theta(n^2)$

Possiamo fare di meglio con la tecnica divide et impera?

Moltiplicazioni di interi con DI

Proviamo a decomporre i due interi x ed y ognuno in due parti aventi lo stesso numero di cifre, idee?

Moltiplicazioni di interi con DI

Proviamo a decomporre i due interi x ed y ognuno in due parti aventi lo stesso numero di cifre, idee?

- $x = x_1 \times B^{\frac{n}{2}} + x_0$
- $y = y_1 \times B^{\frac{n}{2}} + y_0$

Moltiplicazioni di interi con DI

Proviamo a decomporre i due interi x ed y ognuno in due parti aventi lo stesso numero di cifre, idee?

- $x = x_1 \times B^{\frac{n}{2}} + x_0$
- $y = y_1 \times B^{\frac{n}{2}} + y_0$

$$x \times y = \left(x_1 \times B^{\frac{n}{2}} + x_0\right) \times \left(y_1 \times B^{\frac{n}{2}} + y_0\right) =$$

Moltiplicazioni di interi con DI

Proviamo a decomporre i due interi x ed y ognuno in due parti aventi lo stesso numero di cifre, idee?

- $x = x_1 \times B^{\frac{n}{2}} + x_0$
- $y = y_1 \times B^{\frac{n}{2}} + y_0$

$$x \times y = \left(x_1 \times B^{\frac{n}{2}} + x_0\right) \times \left(y_1 \times B^{\frac{n}{2}} + y_0\right) =$$

$$= x_1 \times y_1 \times B^n + x_0 \times y_1 \times B^{\frac{n}{2}} + x_1 \times y_0 \times B^{\frac{n}{2}} + x_0 \times y_0 =$$

Moltiplicazioni di interi con DI

Proviamo a decomporre i due interi x ed y ognuno in due parti aventi lo stesso numero di cifre, idee?

- $x = x_1 \times B^{\frac{n}{2}} + x_0$
- $y = y_1 \times B^{\frac{n}{2}} + y_0$

$$x \times y = \left(x_1 \times B^{\frac{n}{2}} + x_0\right) \times \left(y_1 \times B^{\frac{n}{2}} + y_0\right) =$$

$$= x_1 \times y_1 \times B^n + x_0 \times y_1 \times B^{\frac{n}{2}} + x_1 \times y_0 \times B^{\frac{n}{2}} + x_0 \times y_0 =$$

$$= x_1 \times y_1 \times B^n + (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{2}} + x_0 \times y_0$$

Possiamo fare un algoritmo divide et impera che implementa questa formula? Ci conviene?

Moltiplicazioni di interi con DI

$$= x_1 \times y_1 \times B^n + (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{2}} + x_0 \times y_0$$

Moltiplicazioni di interi con DI

$$= x_1 \times y_1 \times B^n + (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{2}} + x_0 \times y_0$$

- suddividere x in x_1 e x_0 e y in y_1 e y_0, costo max $\theta(n)$
- 4 moltiplicazioni di interi con $\frac{n}{2}$ cifre (ricorsione)
- 3 addizioni di interi con al più $2n$ cifre $\theta(n)$
- 3 moltiplicazioni per B^k, con $k \leq n$. Possono essere fatte semplicemnte aggiungendo k zeri alla fine del numero, costo $\theta(n)$

Moltiplicazioni di interi con DI

$$= x_1 \times y_1 \times B^n + (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{2}} + x_0 \times y_0$$

- suddividere x in x_1 e x_0 e y in y_1 e y_0, costo max $\theta(n)$
- 4 moltiplicazioni di interi con $\frac{n}{2}$ cifre (ricorsione)
- 3 addizioni di interi con al più $2n$ cifre $\theta(n)$
- 3 moltiplicazioni per B^k, con $k \leq n$. Possono essere fatte semplicemnte aggiungendo k zeri alla fine del numero, costo $\theta(n)$

L'algoritmo è di tipo 1 con $a = 4, c = 2, d = 1$

Moltiplicazioni di interi con DI

$$= x_1 \times y_1 \times B^n + (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{2}} + x_0 \times y_0$$

- suddividere x in x_1 e x_0 e y in y_1 e y_0, costo max $\theta(n)$
- 4 moltiplicazioni di interi con $\frac{n}{2}$ cifre (ricorsione)
- 3 addizioni di interi con al più $2n$ cifre $\theta(n)$
- 3 moltiplicazioni per B^k, con $k \leq n$. Possono essere fatte semplicemnte aggiungendo k zeri alla fine del numero, costo $\theta(n)$

L'algoritmo è di tipo 1 con $a = 4, c = 2, d = 1$

Quanto costa?

Moltiplicazioni di interi con DI

$$= x_1 \times y_1 \times B^n + (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{2}} + x_0 \times y_0$$

- suddividere x in x_1 e x_0 e y in y_1 e y_0, costo max $\theta(n)$
- 4 moltiplicazioni di interi con $\frac{n}{2}$ cifre (ricorsione)
- 3 addizioni di interi con al più $2n$ cifre $\theta(n)$
- 3 moltiplicazioni per B^k, con $k \leq n$. Possono essere fatte semplicemnte aggiungendo k zeri alla fine del numero, costo $\theta(n)$

L'algoritmo è di tipo 1 con $a = 4, c = 2, d = 1$

Quanto costa?

$$\frac{a}{c^d} = 2 > 1, \text{ quindi } \theta(n^{\log_c a}) = \theta(n^{\log_2 4}) = \theta(n^2)$$

Non conviene!

Moltiplicazioni di interi con DI

Proviamo a decomporre i due interi x ed y ognuno in tre parti aventi lo stesso numero di cifre.

Moltiplicazioni di interi con DI

Proviamo a decomporre i due interi x ed y ognuno in tre parti aventi lo stesso numero di cifre.

- $x = x_2 \times B^{\frac{2n}{3}} + x_1 \times B^{\frac{n}{3}} + x_0$
- $y = y_2 \times B^{\frac{2n}{3}} + y_1 \times B^{\frac{n}{3}} + y_0$

Moltiplicazioni di interi con DI

Proviamo a decomporre i due interi x ed y ognuno in tre parti aventi lo stesso numero di cifre.

- $x = x_2 \times B^{\frac{2n}{3}} + x_1 \times B^{\frac{n}{3}} + x_0$
- $y = y_2 \times B^{\frac{2n}{3}} + y_1 \times B^{\frac{n}{3}} + y_0$

$$x \times y = \left(x_2 \times B^{\frac{2n}{3}} + x_1 \times B^{\frac{n}{3}} + x_0\right) \times \left(y_2 \times B^{\frac{2n}{3}} + y_1 \times B^{\frac{n}{3}} + y_0\right) =$$

Moltiplicazioni di interi con DI

Proviamo a decomporre i due interi x ed y ognuno in tre parti aventi lo stesso numero di cifre.

- $x = x_2 \times B^{\frac{2n}{3}} + x_1 \times B^{\frac{n}{3}} + x_0$
- $y = y_2 \times B^{\frac{2n}{3}} + y_1 \times B^{\frac{n}{3}} + y_0$

$$x \times y = \left(x_2 \times B^{\frac{2n}{3}} + x_1 \times B^{\frac{n}{3}} + x_0\right) \times \left(y_2 \times B^{\frac{2n}{3}} + y_1 \times B^{\frac{n}{3}} + y_0\right) =$$

$$\begin{aligned} = \quad & x_2 \times y_2 \times B^{\frac{4n}{3}} + (x_2 \times y_1 + x_1 \times y_2) \times B^n + \\ & (x_2 \times y_0 + x_0 \times y_2 + x_1 \times y_1) \times B^{\frac{2n}{3}} + \\ & (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{3}} + x_0 \times y_0 \end{aligned}$$

Moltiplicazioni di interi con DI

Proviamo a decomporre i due interi x ed y ognuno in tre parti aventi lo stesso numero di cifre.

- $x = x_2 \times B^{\frac{2n}{3}} + x_1 \times B^{\frac{n}{3}} + x_0$
- $y = y_2 \times B^{\frac{2n}{3}} + y_1 \times B^{\frac{n}{3}} + y_0$

$$x \times y = \left(x_2 \times B^{\frac{2n}{3}} + x_1 \times B^{\frac{n}{3}} + x_0\right) \times \left(y_2 \times B^{\frac{2n}{3}} + y_1 \times B^{\frac{n}{3}} + y_0\right) =$$

$$\begin{aligned} = \quad & x_2 \times y_2 \times B^{\frac{4n}{3}} + (x_2 \times y_1 + x_1 \times y_2) \times B^n + \\ & (x_2 \times y_0 + x_0 \times y_2 + x_1 \times y_1) \times B^{\frac{2n}{3}} + \\ & (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{3}} + x_0 \times y_0 \end{aligned}$$

- L'algoritmo è di tipo 1 con $a = 9, c = 3, d = 1$

Moltiplicazioni di interi con DI

Proviamo a decomporre i due interi x ed y ognuno in tre parti aventi lo stesso numero di cifre.

- $x = x_2 \times B^{\frac{2n}{3}} + x_1 \times B^{\frac{n}{3}} + x_0$
- $y = y_2 \times B^{\frac{2n}{3}} + y_1 \times B^{\frac{n}{3}} + y_0$

$$x \times y = \left(x_2 \times B^{\frac{2n}{3}} + x_1 \times B^{\frac{n}{3}} + x_0\right) \times \left(y_2 \times B^{\frac{2n}{3}} + y_1 \times B^{\frac{n}{3}} + y_0\right) =$$

$$\begin{aligned} = \quad & x_2 \times y_2 \times B^{\frac{4n}{3}} + (x_2 \times y_1 + x_1 \times y_2) \times B^n + \\ & (x_2 \times y_0 + x_0 \times y_2 + x_1 \times y_1) \times B^{\frac{2n}{3}} + \\ & (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{3}} + x_0 \times y_0 \end{aligned}$$

- L'algoritmo è di tipo 1 con $a = 9, c = 3, d = 1$
- $\frac{a}{c^d} = \frac{9}{3} = 2 > 1$, la complessità è $\theta(n^{\log_c a}) = \theta(n^{\log_3 9}) = \theta(n^2)$, come prima!

Moltiplicazioni di interi con DI

$$x \times y = x_1 \times y_1 \times B^n + (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{2}} + x_0 \times y_0$$

Moltiplicazioni di interi con DI

$$
\begin{aligned}
x \times y &= x_1 \times y_1 \times B^n + (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{2}} + x_0 \times y_0 \\
&= m_3 \times B^n + M \times B^{\frac{n}{2}} + m_1
\end{aligned}
$$

Moltiplicazioni di interi con DI

$$
\begin{aligned}
x \times y &= x_1 \times y_1 \times B^n + (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{2}} + x_0 \times y_0 \\
&= m_3 \times B^n + M \times B^{\frac{n}{2}} + m_1
\end{aligned}
$$

- $m_3 = x_1 \times y_1$ e $m_1 = x_0 \times y_0$

Moltiplicazioni di interi con DI

$$
\begin{aligned}
x \times y &= x_1 \times y_1 \times B^n + (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{2}} + x_0 \times y_0 \\
&= m_3 \times B^n + M \times B^{\frac{n}{2}} + m_1
\end{aligned}
$$

- $m_3 = x_1 \times y_1$ e $m_1 = x_0 \times y_0$
- $M = m_3 + m_2 + m_1 = x_0 \times y_1 + x_1 \times y_0$

Moltiplicazioni di interi con DI

$$
\begin{aligned}
x \times y &= x_1 \times y_1 \times B^n + (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{2}} + x_0 \times y_0 \\
&= m_3 \times B^n + M \times B^{\frac{n}{2}} + m_1
\end{aligned}
$$

- $m_3 = x_1 \times y_1$ e $m_1 = x_0 \times y_0$
- $M = m_3 + m_2 + m_1 = x_0 \times y_1 + x_1 \times y_0$

$$x_1 \times y_1 + m_2 + x_0 \times y_0 = x_0 \times y_1 + x_1 \times y_0$$

Moltiplicazioni di interi con DI

$$x \times y = x_1 \times y_1 \times B^n + (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{2}} + x_0 \times y_0$$

$$= m_3 \times B^n + M \times B^{\frac{n}{2}} + m_1$$

- $m_3 = x_1 \times y_1$ e $m_1 = x_0 \times y_0$
- $M = m_3 + m_2 + m_1 = x_0 \times y_1 + x_1 \times y_0$

$$x_1 \times y_1 + m_2 + x_0 \times y_0 = x_0 \times y_1 + x_1 \times y_0$$

$$m_2 = x_0 \times y_1 + x_1 \times y_0 - x_1 \times y_1 - x_0 \times y_0$$

Moltiplicazioni di interi con DI

$$x \times y = x_1 \times y_1 \times B^n + (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{2}} + x_0 \times y_0$$
$$= m_3 \times B^n + M \times B^{\frac{n}{2}} + m_1$$

- $m_3 = x_1 \times y_1$ e $m_1 = x_0 \times y_0$
- $M = m_3 + m_2 + m_1 = x_0 \times y_1 + x_1 \times y_0$

$$x_1 \times y_1 + m_2 + x_0 \times y_0 = x_0 \times y_1 + x_1 \times y_0$$

$$m_2 = x_0 \times y_1 + x_1 \times y_0 - x_1 \times y_1 - x_0 \times y_0$$

$$m_2 = (x_0 - x_1) \times y_1 + (x_1 - x_0) \times y_0$$

Moltiplicazioni di interi con DI

$$x \times y = x_1 \times y_1 \times B^n + (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{2}} + x_0 \times y_0$$

$$= m_3 \times B^n + M \times B^{\frac{n}{2}} + m_1$$

- $m_3 = x_1 \times y_1$ e $m_1 = x_0 \times y_0$
- $M = m_3 + m_2 + m_1 = x_0 \times y_1 + x_1 \times y_0$

$$x_1 \times y_1 + m_2 + x_0 \times y_0 = x_0 \times y_1 + x_1 \times y_0$$

$$m_2 = x_0 \times y_1 + x_1 \times y_0 - x_1 \times y_1 - x_0 \times y_0$$

$$m_2 = (x_0 - x_1) \times y_1 + (x_1 - x_0) \times y_0$$

$$m_2 = (x_0 - x_1) \times (y_1 - y_0)$$

Moltiplicazioni di interi con DI

$$x \times y = x_1 \times y_1 \times B^n + (x_0 \times y_1 + x_1 \times y_0) \times B^{\frac{n}{2}} + x_0 \times y_0$$
$$= m_3 \times B^n + M \times B^{\frac{n}{2}} + m_1$$

- $m_3 = x_1 \times y_1$ e $m_1 = x_0 \times y_0$
- $M = m_3 + m_2 + m_1 = x_0 \times y_1 + x_1 \times y_0$

$$x_1 \times y_1 + m_2 + x_0 \times y_0 = x_0 \times y_1 + x_1 \times y_0$$
$$m_2 = x_0 \times y_1 + x_1 \times y_0 - x_1 \times y_1 - x_0 \times y_0$$
$$m_2 = (x_0 - x_1) \times y_1 + (x_1 - x_0) \times y_0$$
$$m_2 = (x_0 - x_1) \times (y_1 - y_0)$$

Quindi otteniamo

$$x \times y = m_3 \times B^n + (m_3 + m_2 + m_1) \times B^{\frac{n}{2}} + m_1$$

Moltiplicazioni di interi con DI

Riassumendo abbiamo $x \times y = m_3 \times B^n + (m_3 + m_2 + m_1) \times B^{\frac{n}{2}} + m_1$, con

- $m_3 = x_1 \times y_1$ e $m_1 = x_0 \times y_0$
- $m_2 = (x_0 - x_1) \times (y_1 - y_0)$

Ovverosia un problema DI con

- $a = 3$, $c = 2$ e $d = 1$
- $\frac{a}{c^d} = \frac{3}{2} > 1$ e quindi $T(n) \in \theta(n^{\log_c a}) = \theta(n^{\log_2 3}) == \theta(n^{1.585})$, conviene!

Moltiplicazione di matrici: Algoritmo di Strassen

Possiamo progettare un algoritmo di Divide et Impera basandoci sull'osservazione che una matrice $n \times n$ pu'o essere vista come una matrice 2×2 ove ciascun elemento 'e a sua volta una matrice $n/2 \times n/2$

$$\left[\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right] = \left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \cdot \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] \qquad \begin{cases} r = ae + bg \\ s = af + bh \\ t = ce + dg \\ u = cf + dh \end{cases}$$

Moltiplicazione di matrici: Algoritmo di Strassen

Possiamo progettare un algoritmo di Divide et Impera basandoci sull'osservazione che una matrice $n \times n$ pu'o essere vista come una matrice 2×2 ove ciascun elemento 'e a sua volta una matrice $n/2 \times n/2$

$$\left[\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right] = \left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \cdot \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] \qquad \left\{\begin{array}{lcl} r & = & ae + bg \\ s & = & af + bh \\ t & = & ce + dg \\ u & = & cf + dh \end{array}\right.$$

I parametri dell'algoritmo sarebbero quindi:

- $a = 8$, $c = 2$ e $d = 2$ (dobbiamo eseguire 4 somme di matrici di lato $n/2$)
- $\frac{a}{c^d} = \frac{8}{2^2} = 2 > 1$ e quindi $T(n) \in \theta(n^{\log_c a}) = \theta(n^{\log_2 8}) == \theta(n^3)$, non conviene!

Moltiplicazione di matrici: Algoritmo di Strassen

Strassen ha trovato un metodo per calcolare r, s, t e u con solo 7 moltiplicazioni di matrici $n/2 \times n/2$

$$\text{Ponendo} \quad \begin{cases} P_1 &= a \cdot (f - h) \\ P_2 &= (a + b) \cdot h \\ P_3 &= (c + d) \cdot e \\ P_4 &= d \cdot (g - e) \\ P_5 &= (a + d) \cdot (e + h) \\ P_6 &= (b - d) \cdot (g + h) \\ P_7 &= (a - c) \cdot (e + f) \end{cases} \quad \text{abbiamo che} \quad \begin{cases} r &= P_5 + P_4 - P_2 + P_6 \\ s &= P_1 + P_2 \\ t &= P_3 + P_4 \\ u &= P_5 + P_1 - P_3 - P_7 \end{cases}$$

Moltiplicazione di matrici: Algoritmo di Strassen

Strassen ha trovato un metodo per calcolare r, s, t e u con solo 7 moltiplicazioni di matrici $n/2 \times n/2$

$$\text{Ponendo} \left\{ \begin{array}{lcl} P_1 & = & a \cdot (f - h) \\ P_2 & = & (a + b) \cdot h \\ P_3 & = & (c + d) \cdot e \\ P_4 & = & d \cdot (g - e) \\ P_5 & = & (a + d) \cdot (e + h) \\ P_6 & = & (b - d) \cdot (g + h) \\ P_7 & = & (a - c) \cdot (e + f) \end{array} \right. \quad \text{abbiamo che} \left\{ \begin{array}{lcl} r & = & P_5 + P_4 - P_2 + P_6 \\ s & = & P_1 + P_2 \\ t & = & P_3 + P_4 \\ u & = & P_5 + P_1 - P_3 - P_7 \end{array} \right.$$

I parametri dell'algoritmo sarebbero quindi:

- $a = 7$, $c = 2$ e $d = 2$ (dobbiamo eseguire alcune somme/sottrazioni di matrici di lato $n/2$)
- $\frac{a}{c^d} = \frac{7}{4} => 1$ e quindi $T(n) \in \theta(n^{\log_c a}) = \theta(n^{\log_2 7}) == \theta(n^{2.807})$, conviene!