Quanto costa? **ES5**

```
int n;
cin >> n;
for (int i=0; i<n; ++i)
  for (int j=i; j<n; ++j)
    cout << "(" << i << "," << j << ")\n";
```

La complessità risulta essere **THETA(n^2)**, ma ciò, rispetto a due cicli normali (ES 4) con i=j=0 in partenza, non deriva da una moltiplicazione "pura" n*n = n^2 ma bensì dallo sviluppo (dimostrato per induzione) della sommatoria da k=0 a n di k (= [n(n+1)]/2) che sviluppata presenta il termine dominante n^2. Svolgiamo infatti n+(n-1)+(n-2)+...1 cicli.

Quanto costa? **ES6**

```
void create_empty(basic_list& list)
{
  cell* aux = new cell;
  aux->next = aux;
  list = aux;
}
```

La complessità risulta **THETA(1)** in maniera triviale.

Quanto costa? **ES7**

```
void head_insert(basic_list& list, DataType new_value) {
  cell* aux = new cell;
  aux->payload = new_value;
  cell* tmp = list->next;
  list->next = aux;
  aux->next = tmp;
}
```

La complessità risulta **THETA(1)** in maniera triviale.

Quanto costa? **ES8**

```
void print_list(std::ostream& output_stream, basic_list list)
{
    cell* aux = list->next; // devo saltare la sentinella
    while (aux != list) {
        WriteData(output_stream, aux->payload);
        aux = aux->next;
        output_stream << std::endl;
    }
}
```

Stampare una lista, in questo caso circolare come si evince da aux != list, comporta ovviamente stampare i suoi n elementi (considerando costante WriteData). **THETA(n)**.

Quanto costa? **ES9**

```
void read_list(std::istream& input_stream, basic_list& list)
{
    create_empty(list);
    DataType d;
    while (ReadData(input_stream, d)) {
        head_insert(list, d);
    }
}
```

create_empty e ReadData hanno THETA(1), dunque la complessità è data dalla iterazione su ReadData degli n elementi presenti nello stream, che vengono inseriti in list. **THETA(n)**.

Quanto costa? **ES10**

```
void read_list(std::istream& input_stream, basic_list& list)
{
    create_empty(list);
    DataType d;
    while (ReadData(input_stream, d)) {
        last_insert(list, d); // supponendo che last_insert
inserisca un elemento in coda alla lista list, che è una
lista collegata semplicemente, circolare e con sentinella
    }
}
```

Rispetto ad ES9, devo inserire in CODA e non in TESTA. Dato che non mi trovo su una lista doppiamente collegata circolare con sentinella, ma la lista è SEMPLICEMENTE collegata; non posso sfruttare il puntatore a prev per inserire in coda con THETA(1), ma dovrò inserire

proporzionalmente, man mano, agli elementi di n già inseriti (devo scorrere la lista) → risulta una sommatoria da k=0 a n di k (= [n(n+1)]/2) che sviluppata presenta il termine dominante n^2. La complessità risulta dunque essere in conclusione **THETA(n^2)**.

Quanto costa? **ES11**

```
double fact(int n)
{
if (n==0)
   return 1;
else
  return n*fact(n-1);
}
```

Svolgo un'operazione di moltiplicazione per n volte. La complessità risulta quindi banalmente **THETA(n)**. Tuttavia in questo caso risulta utile annotare che, in caso di numeri grandi, il fattoriale potrebbe diventare "ingestibile". Non ce ne occupiamo.

Quanto costa? **ES12**

```
int prodRecAux(const Seq& l){
   if (isEmpty(l)) return 1;
   else return l->head * prodRecAux(l->tail);
}


int prodRec(const Seq& l){
   if (isEmpty(l)) return -1;
   else return prodRecAux(l);
}
```

La complessità della funzione non deve spaventare. Ci interessa solo che venga effettuata n volte la moltiplicazione su tutti gli elementi di l (produttoria). Ergo **THETA(n)**.

Dovevo riempire questo spazio vuoto, mi dava fastido.

Quanto costa? **ES13**

```
bool isThereLesserThanRec(const Elem elem, const
Seq& l){
  if (isEmpty(l)) return false;       // se l'insieme e' vuoto
restituisce false
  if (l->head < elem) return true;
  else return isThereLesserThanRec(elem, l->tail);
}
```

Incontriamo un algoritmo con caso migliore e caso peggiore. Questo perché se già il primo elemento della lista è minore di elem, ritorno true e l'algoritmo termina con **THETA(1)**.
Se invece tutti gli elementi risultano essere maggiori di elem (non esiste x<elem) allora avrò come caso peggiore **THETA(n)** dopo aver scandito tutta la lista. Dunque il caso migliore corrisponde al ritorno di true, mentre il caso peggiore al ritorno di false (arrivo ad isEmpty).

Quanto costa? **ES14**

```
bool areAllGreaterEqThan(const Elem elem, const Seq&
l){
  return (!isThereLesserThan(elem, l)); // se l'insieme e'
vuoto restituisce true
}
```

Si rimanda all'ES 13. Avremo dunque caso migliore **THETA(1)**, ma questa volta quando il return finale è un false (opposto di ES13), e caso peggiore **THETA(n)** quando il return finale è un true. La funzione, come già visto ad IP e rifacendosi al corso di Logica, sfrutta le leggi di DeMorgan per i quantificatori (esistenziale ed universale).

Quanto costa? **ES15**

```
bool member(const Elem e, const Seq& s)
// se si sa che la sequenza è implementata mediante una lista
ordinata, e' possibile ottimizzare il codice; in questo caso non
facciamo alcuna assunzione
{
  if (isEmpty(s))  // se la sequenza e' vuota l'elemento non c'e':
restituisco false
    return false;

  if (s->head == e)  // se la head e' l'elemento cercato restituisco
true
    return true;
  else
    return member(e, s->tail); // altrimenti chiamo ricorsivamente
sulla coda, restituendo il risultato della chiamata ricorsiva
}
```

Anche in questo caso abbiamo un algoritmo ADATTIVO. Per la precisione nel caso migliore (l'elemento che stiamo cercando è il primo) avremo complessità **THETA(1)**, nel caso peggiore dobbiamo ovviamente scansionare la lista, ottenendo **THETA(n)**.

Quanto costa? **ES16**

```
Error insertElem(const Elem x, Seq& s)
// insertElem aggiunge sempre in prima posizione; non rispetta quindi l'ordine
"temporale" con cui gli elementi vengono inseriti, ma e' molto piu' efficiente
rispetto a un inserimento in coda; se stiamo implementando un insieme, in cui la
posizione degli elementi non conta, va benissimo inserire nel punto che rende
l'operazione piu' efficiente
{
   if (member(x, s))
      return PRESENT; // l'elemento e' gia' presente, non lo aggiungo

   cell *aux;
   try  { aux = new cell; }
   catch(...) { return FAIL;} //heap esaurito
   aux->head=x;
   aux->tail=s;
   s = aux;
   return OK;
}
```

Tutto ciò che accade dopo ha complessità costante, dunque tutto dipende dalla funzione member (descritta nell'ES15) → **THETA(1)** nel caso migliore e **THETA(n)** nel peggiore.

Quanto costa? **ES17**

```
Seq seqReadFromStream(istream& str)
{
   Seq s = EMPTY_SEQ;
   Elem e;
   str>>e;
   if (!str) throw runtime_error("Errore inserimento dati\n");
   while (e!= FINEINPUT)  // assumiamo che il segnale di fine input nel file sia il
numero  FINEINPUT
      {
      insertElem(e, s);
      str>>e;
      if (!str) throw runtime_error("Errore inserimento dati\n");
      }
return s;
}
```

Partiamo con l'individuare gli unici due nodi importanti per il nostro calcolo: il ciclo sugli elementi letti e la funzione insertElem (ES16). Sapendo che la funzione insertElem ha complessità THETA(1) nel caso migliore, ne deduciamo che di conseguenza (ripetendo questa operazione n volte per via del ciclo) la complessità nel caso migliore è **THETA(n)**. Invece, nel caso peggiore, inseriamo gli elementi mano a mano, proporzionalmente alla lunghezza della lista che cresce ad ogni inserimento. Deduciamo dunque una sommatoria, da cui → **THETA(n^2)** nel caso peggiore.

Quanto costa? **ES18**

Inserimento in coda in una lista semplice circolare con sentinella

Abbiamo **THETA(n)** (scansione della lista per inserire in coda, non avendo ->prev).

Quanto costa? **ES19**

Inserimento in coda in una lista doppiamente collegata circolare con sentinella

Siamo giunti al caso tanto citato. La complessità è **THETA(1)** perchè possiamo balzare direttamente all'ultimo elemento, un po' come il gatto della prof.Mascardi quando la cena è pronta. Okay, questo era cringe, ma sto scrivendo da 3 ore e sono stanco.

Quanto costa?

- **ES20:** dato un array **A** non ordinato*, verificare se l'elemento **e** appartiene all'array
- **ES21:** dati due array **A** e **B** non ordinati, verificare se un elemento **e** appartiene a uno dei due
- **ES22:** dati due array **A** e **B** non ordinati, verificare se hanno un elemento in comune
- **ES23:** dato un array **A** non ordinato, verificare se presenta elementi duplicati
- **ES24:** dato un array **A** ordinato, verificare se l'elemento **e** appartiene all'array

ES20: In modo triviale, se troviamo subito l'elemento: **THETA(1)** (caso migliore). Se invece l'elemento non è presente abbiamo **THETA(n)** (caso peggiore).
ES21: Se troviamo subito l'elemento (1a posizione del 1o array): **THETA(1)** (caso migliore). Se invece l'elemento non è presente in nessuno dei due array, devo scandirli entrambi, ottenendo una complessità in funzione sia di n sia di m (due dimensioni dei due array differenti; non ipotizzando che abbiano la stessa dim.) → **THETA(n+m)** (caso peggiore).
ES22: Se troviamo subito l'elemento in comune tra i due array, al primo controllo incrociato: **THETA(1)** (caso migliore). Se invece i due array (dim n ed m) non hanno elementi in comune, li devo scorrere entrambi. Si ipotizza un algoritmo "dummy" in cui sono semplicemente presenti due cicli annidati che scandiscono → **THETA(n*m)** (caso peggiore).
ES23: Se troviamo subito l'elemento duplicato: **THETA(1)** (caso migliore). Se invece l'elemento non è presente abbiamo **THETA(n^2)** (caso peggiore). Questo n^2 può derivare da due cicli annidati (controllo ogni elemento per ogni elemento) → n*n ; oppure da un

algoritmo più “furbo” in cui a ogni iterazione esterna escludo il sottoarray di sinistra dalla ricerca (di sicuro lì non c’è un duplicato: pensaci!) → in questo caso avrò una sommatoria, che pur essendo migliore di un puro n*n degenera in THETA(n^2). Purtroppo, è la vita.
ES24: Non sappiamo ancora farlo, ma Patrick dice che la complessità della ricerca binaria è logaritmica. Per ora non ci fidiamo.

Se ce la fa il gatto, puoi farcela anche te. Tieni duro bro.

```
void selectionSort(vector<int>& v)
{
   int current_min_index;
   unsigned int size = v.size();
   for (unsigned int i=0; i<size; ++i)
   {
     current_min_index = i;
     for (unsigned int j=i+1;j<size; ++j)
        if (v[current_min_index] > v[j])
            current_min_index = j;
     scambia(v, i, current_min_index);
   }
}
```

Il selection sort non è adattivo. Ne risulta che in ogni caso la complessità è **THETA(n^2)**. Questo perchè non essendoci un’ottimizzazione nel caso di sottosequenze già ordinate, il selection effettua sempre e comunque tutte le iterazioni dei cicli. (deriva da sommatoria)

```
void insertionSort(vector<int>& v)
{
   int current, prev;
   unsigned int size = v.size();
   for (unsigned int i=1; i<size; ++i)
   { current=i;
     prev=i-1;
     while(prev>=0 && v[current]<v[prev])
        {
         scambia(v, current, prev);
         --current;
         --prev;
        }
   }
}
```

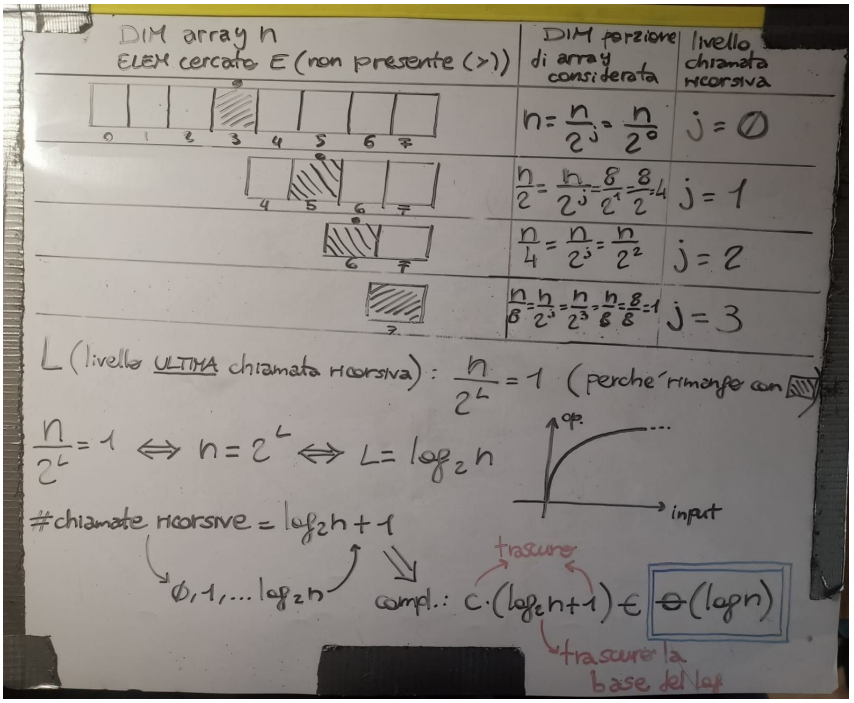
Laddove il caso peggiore, come nel selectionSort da sommatoria, risulta **THETA(n^2)**, in questo caso l’insertionSort è adattivo, e dunque si ha un caso migliore con **THETA(n)**; questo per costruzione interna dell’algoritmo (il confronto a sinistra, se “fallisce” fa fermare l’iterazione: non è necessario che vengano effettuati ulteriori confronti → n confronti e nessuno scambio). Il selection sort effettuava comunque sia confronti sia scambi.

```
void bubbleSort(vector<int>& v)
{
   unsigned int size = v.size();
   bool scambiati;
   for (unsigned int i=1; i<size; ++i)
      {
      scambiati = false;
      for (unsigned int j=0; j<size-i; ++j)
          if(v[j]>v[j+1])
            {
              scambia(v, j, j+1);
              scambiati = true;
            }
      if (!scambiati) return;
      }
}
```

Il bubble sort classico avrebbe come UNICA complessità THETA(n^2), ma grazie all'utilizzo del flag booleano "scambiati" otteniamo come caso migliore **THETA(n)**: se l'array è già ordinati infatti alla prima iterazione esterna completa (n) il flag sarà ancora false perchè nessun confronto sarà stato effettuato, portando il programma alla terminazione. Il caso peggiore, in maniera triviale, rimane **THETA(n^2)** come gli altri due algormti quadratici.

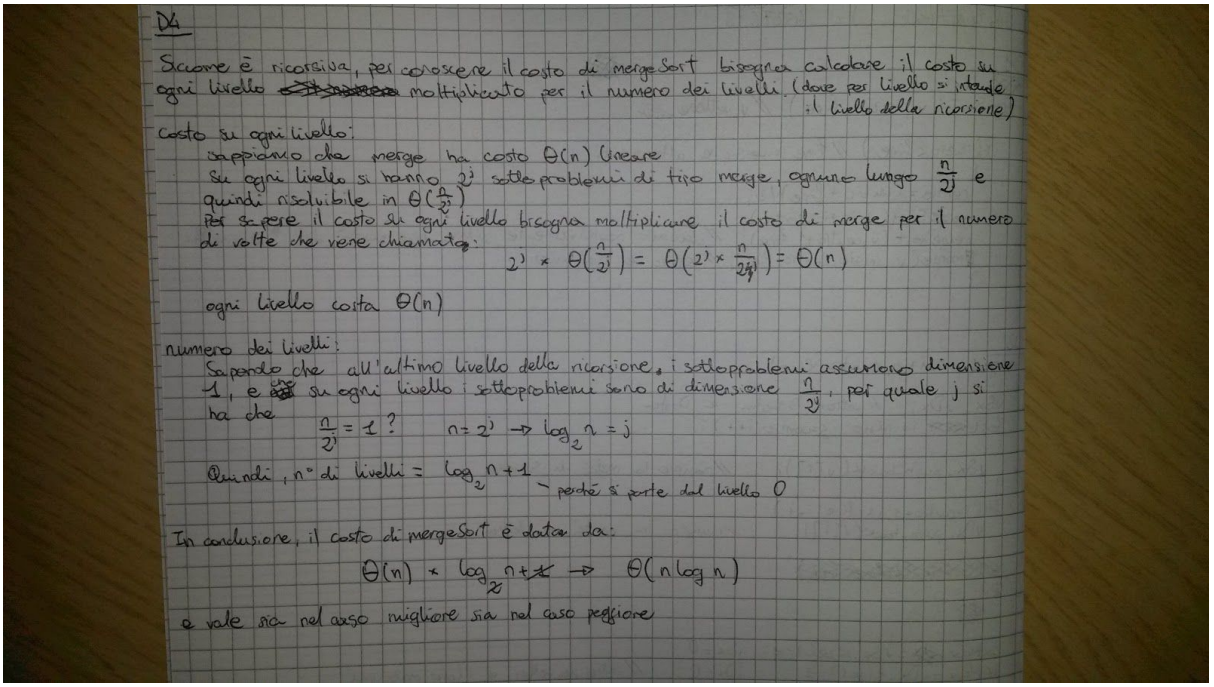
La ricerca binaria

```
int ricercaBinariaAux(int inizio, int fine, int array[], int elem)
{
if (inizio==fine)
   {
   if (array[inizio]==elem)
      return inizio;
   else
      return -1;
   }

int mezzo = (inizio+fine)/2;
if (array[mezzo]==elem) return mezzo;

if (elem > array[mezzo])
   return ricercaBinariaAux(mezzo+1,fine,array,elem);
else
   return ricercaBinariaAux(inizio,mezzo-1,array,elem);
}
```



MERGE SORT

Non metto il codice perchè è troppo lungo



Q4

Siccome è ricorsiva, per conoscere il costo di mergeSort bisogna calcolare il costo su ogni livello ~~[illegible]~~ moltiplicato per il numero dei livelli (dove per livello si intende il livello della ricorsione)

Costo su ogni livello:
sappiamo che merge ha costo $\Theta(n)$ lineare
Su ogni livello si hanno 2^j sottoproblemi di tipo merge, ognuno lungo $\frac{n}{2^j}$ e quindi risolvibile in $\Theta(\frac{n}{2^j})$
Per sapere il costo su ogni livello bisogna moltiplicare il costo di merge per il numero di volte che viene chiamato:

$$2^j \times \Theta\left(\frac{n}{2^j}\right) = \Theta\left(2^j \times \frac{n}{2^j}\right) = \Theta(n)$$

ogni livello costa $\Theta(n)$

numero dei livelli:
Sapendo che all'ultimo livello della ricorsione, i sottoproblemi assumono dimensione 1, e su ogni livello i sottoproblemi sono di dimensione $\frac{n}{2^j}$, per quale j si ha che

$$\frac{n}{2^j} = 1 ? \quad n = 2^j \rightarrow \log_2 n = j$$

Quindi, n° di livelli $= \log_2 n + 1$ — perché si parte dal livello 0

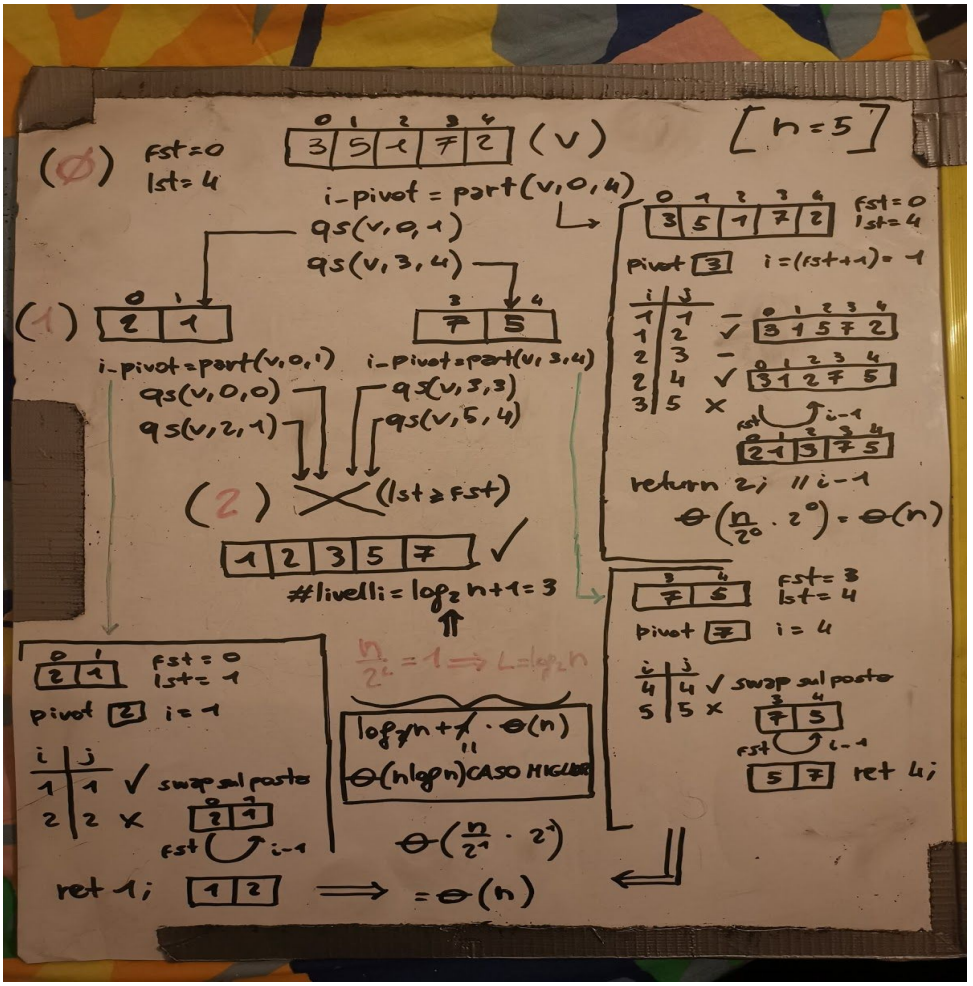
In conclusione, il costo di mergeSort è dato da:

$$\Theta(n) \times \log_2 n \rightarrow \Theta(n \log n)$$

e vale sia nel caso migliore sia nel caso peggiore

GIUSTA ma migliorabile, come?

QUICK SORT



Si fa riferimento ad un quick sort con algoritmo IN PLACE.
Nella foto sopra osserviamo la complessità nel caso migliore di QUICK SORT, ossia quello in cui “magicamente” vengono scelti come PIVOT sempre gli elementi mediani (ossia gli elementi che, se ordinassimo la sequenza, risulterebbero nella posizione n/2 (+/- se dispari)). In questo specifico caso, essendo gli elementi pochissimi, basta che solo la prima volta venga scelto il mediano, e nei casi successivi sarà triviale (su 2 elementi, per forza, viene scelto il mediano). La sequenza è stata dunque divisa a metà ad ogni livello dell’albero della ricorsione, ottenendo un calcolo della complessità logicamente equivalente a quello del Merge Sort (nel suo unico caso, in quanto algoritmo non adattivo): **THETA(n*log(n)).**
Il caso medio non è stato calcolato, perchè richiederebbe conoscenze statistiche e probabilistiche. La sua complessità permane **THETA(n*log(n))** (ecco evidenti i limiti della notazione THETA!).
Prima di riportare la complessità del caso peggiore, consideriamo una metodologia di scelta del pivot diversa da quella banale (prima posizione), ossia la scelta randomica del pivot. Importante dunque notare che la scelta del pivot non è direttamente collegata alla complessità che otterremo, anche se può ovviamente influenzarla positivamente o meno.
Scegliendo il pivot a caso tra gli elementi disponibili, si spera di sceglierlo “abbastanza buono” “abbastanza spesso”
● *Abbastanza buono: un pivot che partizioni l'array in modo tale che 25% degli elementi sia < del pivot e 75% sia >=, è sufficiente per restare nella complessità Θ(n log n) [non lo dimostriamo]*
● *Abbastanza spesso: metà degli elementi dà una partizione 25%-75% o anche migliore*
Riportando ora la complessità nel caso peggiore (**THETA(n^2)**) ci accorgiamo che Quick Sort può apparentemente essere “peggiore” di Merge, ma per un insieme di considerazioni statistiche e spaziali (Merge usa strutture d’appoggio), in realtà non è così!
La complessità di n^2 si può ottenere, in generale, se ad ogni chiamata viene scelto come pivot il primo o ultimo elemento della sequenza, andando a generare due chiamate ricorsive, di cui una su 0 elementi (nulla) e un’altra su n-1 (si vanno così a perdere le agognate proprietà del Divide Et Impera) → otteniamo dunque una sommatoria da 1 ad n che ci riconduce ad una complessità quadratica!
Un ESEMPIO può essere un QuickSort con scelta banale del Pivot chiamato su un array già ordinato.