Esempio 8.1. Dato un array A di n interi compresi tra 1 e k:

$$\forall 0 < j \leq n. A[j] \in [1, \ldots, k]$$

```
linearSort(A:[Int], B:[Int], k:Int) -> Void {
  // Inizializzo un array che tiene conto dei numeri da 1 a k
  for (var i:Int = 1; i<=k; i++) C[i] = 0;  Θ(k)

  var j:Int = 1;
  // Conto quante volte compare ogni numero nell'array originale
  for (j=1; j<=n; j++) C[A[j]] += 1;  Θ(n)

  j=1;
  var z:Int = 1;
  // Dispongo ogni numero nell'array finale in ordine sapendo quante volte compare
  for (z=1; z <= k; z++) {  Θ(k)
    for (var v:Int = 0; v < C[z]; v++) {  Θ(n)
      B[j] = z;
      j++;
    }
  }
}
```

Listing 15: Algoritmo linear sort

Complessità
In questo caso la complessità è $\Theta(n + k)$ e si usa quando $k \in O(n)$.

8.3.1 Radix sort

Questo algoritmo funziona in maniera simile a come il cervello umano ordina gruppi di numeri: si ordinano (tramite un algoritmo di ordinamento **stabile** prima le cifre delle migliaia, poi quelle delle centinaia, quelle delle decine ed infine le unità. Notiamo però che il risultato NON è corretto.

1094	**9**86	10**9**4	12**5**	1120
986	**2**34	1**2**5	112**0**	234
234	**1**25	11**2**0	23**4**	1094
125	1**0**94	2**3**4	98**6**	125
1120	1**1**20	9**8**6	109**4**	986

Per farlo funzionare dobbiamo ordinare le cifre partendo da quelle meno significative, quindi dalle unità.

109**4**	11**2**0	1**1**20	**1**094	125
98**6**	10**9**4	**1**25	**1**120	234
23**4**	2**3**4	**2**34	125	986
12**5**	1**2**5	**9**86	234	1094
112**0**	9**8**6	1**0**94	986	1120

9 Complessità

9.1 Notazione asintotica

Quando scriviamo un algoritmo, per calcolarne il costo, bisogna fare una serie di assunzioni sulla macchina astratta su cui lavoriamo:

- L'accesso alle celle di memoria avviene in tempo costante.
- Le operazioni elementari avvengono in tempo costante:
  - Operazioni aritmetiche e logiche della ALU

- Gli assegnamenti
- I controlli del flusso (salti, assegnamento al registro PC)

Per calcolare il costo degli algoritmi si possono utilizzare due modelli:

1. **Word model**: tutti i dati occupano solo una cella di memoria.
2. **Bit model**: unità elementare di memoria *bit*, si usa quando le grandezze sono troppo grandi.

Esistono una serie di parametri da **analizzare** quando scriviamo una algoritmo. Essi permettono di garantire il suo corretto funzionamento e la sua ottimizzazione. Sono i seguenti:

- **Complessità**: ovvero l'analisi dell'utilizzo delle risorse:
  - tempo di esecuzione
  - spazio di memoria per i dati in ingresso e in uscita. Viene rappresentato astrattamente dal numero di celle di memoria (word model)
  - banda di comunicazione (per esempio nel caso il calcolo sia distribuito)

  Non sarà quasi mai possibile avere un programma che è sia efficiente in termini di tempo che in di spazio (*coperta corta*).
- **Correttezza**: Indica se l'algoritmo fa quello per cui è stato progettato. Si esegue in due modi:
  - dimostrazione formale la quale permette di dimostrare la correttezza risolvendo tutte le istanze del problema
  - ispezione formale nella quale si usano metodi come il **testing** o il **profiling**. Il primo prevede di provare il programma nelle situazioni critiche, il secondo analizza il tempo che la CPU impiega per elaborare una determinata parte del programma.
- **Semplicità**: Indica se l'algoritmo è facile da capire e manutenere. Un algoritmo è semplice quando usa identificatori significativi, quando è ben commentato, se usa strutture dati adeguate e se rispetta gli standard.

Definizione 9.1 (Complessità di un problema)**.** *La complessità di un problema P è la complessità del miglior algoritmo A che lo risolve.*

Per trovare la complessità del problema partiamo dal fatto che, dato un algoritmo A, la complessità di A determina un limite superiore alla complessità di P (cioè quando si verifica il caso peggiore uso A per risolvere P).
Se riusciamo a determinare un limite inferiore $g(n)$ per P, per ogni algoritmo A che risolve P ho che $A \in \Omega(g(n))$, dove $g(n)$ è il minimo numero di operazione che posso impiegare per risolvere P. Quindi possiamo dire che:

$$A \in \Theta(g(n)) \implies A \text{ ottimo}^3 \tag{16}$$

Per fare ciò bisogna anche andare a calcolare il limite inferiore del caso pessimo, e ciò è possibile tramite 3 metodi: la **dimensione dei dati**, gli **eventi contabili** e gli **alberi decisionali**.

- **Dimensione dei dati**: Se la soluzione di un problema richiede l'esame di tutti i dati in input, allora $\Omega(n)$ è un limite inferiore. *E.g. sommare tutti gli elementi di un array.*
- **Eventi contabili**: se la soluzione di un problema richiede la ripetizione di un certo evento, allora il numero di volte che l'evento si ripete (moltiplicato per il suo costo) è un limite inferiore.
- **Alberi di decisione**: sono alberi in cui
  - ogni nodo non foglia effettua un test su un attributo
  - ogni arco uscente da un nodo è un possibile valore dell'attributo
  - ogni nodo foglia assegna una classificazione

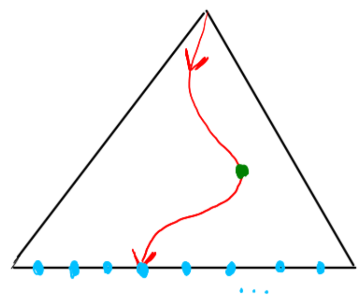

Figure 9: Albero decisionale

Si applica a problemi risolubili attraverso sequenze di decisioni che via via riducono lo spazio delle soluzioni.
In figura vediamo che dalla situazione iniziale, tramite un percorso radice-foglia (ovvero un'esecuzione dell'algoritmo), otteniamo una tra le possibili soluzioni (foglie) passando per diverse decisioni (nodi interni).

Note 9.1.1. *Alcune formule importanti per gli alberi:*

- ***Foglie:*** n^d
- ***Profondità*** $d \leq \log_n$ *foglie (è esattamente uguale solo se l'albero è completo)*
- ***Nodi:*** $n^{d+1} - 1$

Esempio 9.1. Ricerca binaria di un elemento k in un array A di n elementi. Ogni confronto tra k e $A[cen]$ può generare 3 possibili risposte:

- $k < A[cen]$ ramo sinistro
- $k == A[cen]$ ramo centrale
- $k > A[cen]$ ramo destro

Abbiamo quindi che ogni confronto apre 3 possibili vie e dopo i confronti avremo 3^i vie. Le possibili soluzioni sono $n + 1$ (k può essere in ognuna delle n posizioni o non esserci). Avremo quindi:

$$3^i \geq n + 1 > n \implies \text{binSearch} \in \Omega(\log_n) \tag{17}$$

9.2 Big-O notation

La notazione Big-O ha molteplici scopi nella scrittura di un algoritmo.

- Serve a rappresentare la complessità relativa di un algoritmo.
- Descrive le prestazioni di un algoritmo e come queste scalano al cresce dei dati in input.
- Descrive un limite superiore al tasso di crescita di una funzione ed è il caso peggiore.

9.2.1 Limite superiore asintotico

Definizione 9.2 (Limite superiore asintotico)**.** *Il limite superiore asintotico* [4] *si definisce come:*

$$O(g(n)) = \{f(n) \mid \exists c, n_0 > 0. \forall n > n_0, 0 \leq f(n) \leq c \cdot g(n)\} \tag{18}$$

Si scrive come $f(n) \in O(g(n))$ oppure $f(n) = O(g(n))$ e si legge $f(n)$ è nell'ordine O grande di $g(n)$.

Esempio 9.2. Esempio di calcolo del limite superiore asintotico

[3] Ricorda che dire che $A \in \Theta(g(n))$ vuol dire che $A \in O(g(n))$ e $A \in \Omega(g(n))$

[4] *Asintotico* indica che la definizione deve essere valida solo da un certo punto in poi scelto arbitrariamente.

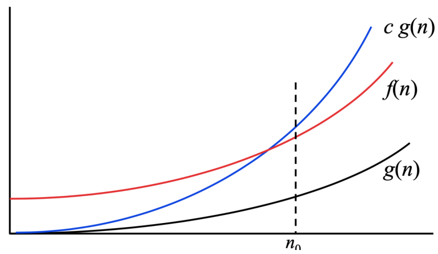

Figure 10: Limite superiore asintotico

Prendiamo due funzioni e determiniamo i punti n_0 e c per cui è soddisfatta la definizione.

$$f(n) = 3n^2 + 5 \qquad g(x) = n^2$$

Stabiliamo un $c = 4$ e $n_0 = 3$.

1. $4 \cdot g(n) = 4n^2 = 3n^2 + n^2$
2. $3n^2 + n^2 \geq 3n^2 + 9$ (per ogni $n \geq 3$)
3. $3n^2 + 9 > 3n^2 + 5 \Longrightarrow 4 \cdot g(n) > f(n)$

Note 9.2.1. *Abbiamo disegnato solo il primo quadrante perché sia i dati in input che le operazioni da eseguire saranno sempre in numero positivo.*

9.2.2 Limite inferiore asintotico

Definizione 9.3 (Limite inferiore asintotico)**.** *Il limite inferiore asintotico si definisce come:*

$$\Omega(g(n)) = \{f(n) \mid \exists c, n_0 > 0. \forall n > n_0, 0 \leq c \cdot g(n) \leq f(n)\} \tag{19}$$

Si scrive come $f(n) \in \Omega(g(n))$ oppure $f(n) = \Omega(g(n))$ e si legge $f(n)$ è nell'ordine Ω grande di $g(n)$. Indica che quell'algoritmo non potrà mai fare di meglio.

Esempio 9.3. Esempio di calcolo del limite inferiore asintotico. Prendiamo due funzioni e determiniamo i punti n_0 e c per cui è soddisfatta la definizione.

$f(n) = \frac{n^2}{2} - 7 \qquad g(x) = n^2$
Stabiliamo un $c = \frac{1}{4}$ e $n_0 = 6$.

1. $\frac{1}{4} \cdot g(n) = \frac{n^2}{4} = \frac{n^2}{2} - \frac{n^2}{4}$
2. $\frac{n^2}{2} - \frac{n^2}{4} \leq \frac{n^2}{2} - 9$ (per ogni $n \geq 6$)
3. $\frac{n^2}{2} - 9 > \frac{n^2}{2} - 7 \Longrightarrow \frac{1}{4} \cdot g(n) < f(n)$

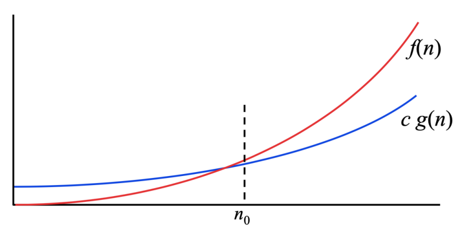

Figure 11: Limite superiore asintotico

9.2.3 Limite asintotico stretto

Definizione 9.4 (Limite asintotico stretto)**.** *Il limite asintotico stretto si definisce come:*

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 > 0. \forall n > n_0, 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\} \tag{20}$$

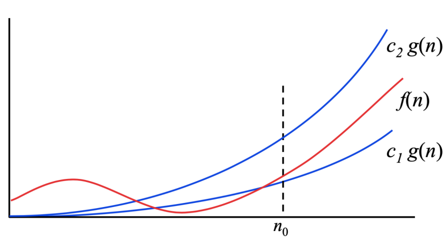

Figure 12: Limite asintotico stretto

Si scrive come $f(n) \in \Theta(g(n))$ oppure $f(n) = \Theta(g(n))$ e si legge $f(n)$ è nell'ordine Θ grande di $g(n)$.

Dalla definizione deriva che:

$$f(n) \in \Theta(g(n)) \Longleftrightarrow f(n) \in \Omega(g(n)) \wedge f(n) \in O(g(n)) \tag{21}$$

9.2.4 Teoremi sulla notazione asintotica

Teorema 9.1. *Per ogni $f(n)$ e $g(n)$ vale che:*

1. $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$
2. *Se* $f_1(n) = O(f_2(n)) \land f_2(n) = O(f_3(n)) \implies f_1(n) = O(f_3(n))$
3. *Se* $f_1(n) = \Omega(f_2(n)) \land f_2(n) = \Omega(f_3(n)) \implies f_1(n) = \Omega(f_3(n))$
4. *Se* $f_1(n) = \Theta(f_2(n)) \land f_2(n) = \Theta(f_3(n)) \implies f_1(n) = \Theta(f_3(n))$
5. *Se* $f_1(n) = O(g_1(n)) \land f_2(n) = O(g_2(n)) \implies O(f_1(n) + f_2(n)) = O(max\{g_1(n), g_2(n)\})$
6. *Se* $f(n)$ *è un polinomio di grado* $d \implies f(n) = \Theta(n^d)$

9.2.5 Limite superiore asintotico non stretto

Definizione 9.5 (Limite superiore asintotico non stretto)**.** *Il limite superiore asintotico non stretto si definisce come:*

$$o(g(n)) = \{f(n) \mid \forall c \exists n_0 > 0. \forall n > n_0, 0 \leq f(n) \leq c \cdot g(n)\} \tag{22}$$

Si scrive come $f(n) \in o(g(n))$ oppure $f(n) = o(g(n))$ e si legge $f(n)$ è nell'ordine o piccolo di $g(n)$. $f(n)$ è limitata superiormente da $g(n)$, ma non la raggiunge mai.

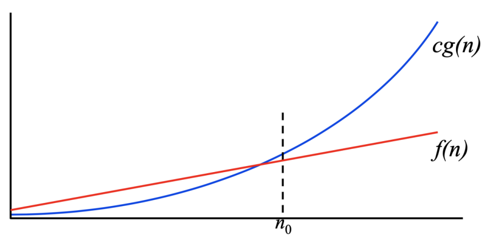

Figure 13: Limite superiore non stretto

E immediato dalla definizione che:

$$o(g(n)) \implies O(g(n))$$

Non vale il contrario:

$$2n^2 \in O(n^2) \land 2n^2 \notin o(n^2)$$

Definizione alternativa:

$$f(n) \in o(g(n)) \iff \lim_{n \to \infty} \frac{g(n)}{f(n)} = \infty$$

9.2.6 Limite inferiore asintotico non stretto

Definizione 9.6 (Limite inferiore asintotico non stretto)**.** *Il limite inferiore asintotico non stretto si definisce come:*

$$\omega(g(n)) = \{f(n) \mid \forall c\ \exists n_0 > 0. \forall n > n_0, 0 \leq c \cdot g(n) \leq f(n)\}$$

Si scrive come $f(n) \in \omega(g(n))$ oppure $f(n) = \omega(g(n))$ e si legge $f(n)$ è nell'ordine ω piccolo di $g(n)$. $f(n)$ è limitata inferiormente da $g(n)$, ma non la raggiunge mai.

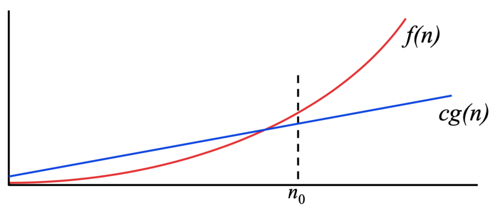

Figure 14: Limite asintotico stretto

E immediato dalla definizione che:

$$\omega(g(n)) \implies \Omega(g(n))$$

Non vale il viceversa:

$$\frac{1}{5}n^2 \in \Omega(n^2) \land \frac{1}{2}n^2 \notin \omega(n^2)$$

Definizione alternativa:

$$f(n) \in \omega(g(n)) \iff \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

9.3 Equazioni di ricorrenza

Quando un algoritmo contiene una chiamata *ricorsiva* a se stesso, il suo tempo di esecuzione può essere descritto da una **equazione di ricorrenza**.

Definizione 9.7 (Ricorrenza). *Una ricorrenza è un'equazione o una disequazione che descrive una funzione in termini del suo valore su input sempre più piccoli.*
*Un'**equazione ricorsiva** esprime il valore di $T(n)$ come combinazione di $T(n_1), \dots, T(n_h)$ dove $n_i < n, i = 1, \dots, h$:*

$$T(n) = \begin{cases} c & n \leq k \\ D(n) + \sum_{i=1}^{h} T(n_i) + C(n) & n > k \end{cases} \tag{23}$$

In un'equazione di ricorrenza:

- $\mathbf{T(n)}$ è il *tempo di esecuzione* di un problema di dimensione n
- Suddividiamo un problema in a **sotto problemi**, ciascuno di dimensione n/b:
  - $\mathbf{D(n)}$ è il tempo impiegato per *suddividere*
  - $\mathbf{C(n)}$ per *combinare* le soluzioni

$$T(n) = \begin{cases} \Theta(1) & n \leq k \\ a \cdot T(\frac{n}{b}) + D(n) + C(n)) & n > k \end{cases} \tag{24}$$

Alcuni casi particolari di equazioni ricorrenza sono quelle di **ordine k**:

$$T(n) = \begin{cases} \Theta(1) & n \leq k \\ \alpha_1 \cdot T(n-1) + \ldots + \alpha_k \cdot T(n-k) + f(n) & n > k \end{cases} \tag{25}$$

e quelle **bilanciate**:

$$T(n) = \begin{cases} \Theta(1) & n \leq k \\ a \cdot T(\frac{n}{b}) + f(n) & n > k \end{cases} \tag{26}$$

con $a \geq 1$, $b > 1$, f asintoticamente positiva.

9.3.1 Metodo iterativo

Prendiamo come esempio l'algoritmo di ordinamento *merge sort* e analizziamone la complessità:

```
void sort(int a[], size_t inizio, size_t fine, char order) {
  if ((fine - inizio) >= 1) {
    size_t centro1 = (inizio + fine)/2; Θ(1)
    zie_t centro2 = centro1 + 1; Θ(1)

    sort(a, inizio, centro1, order); T()
    sort(a, centro2, fine, order);

    merge(a, inizio, centro1, centro2, fine, order);
  }
}
```

Listing 16: Algoritmo merge sort

Ora scriviamo l'equazione di ricorrenza come segue:

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ 2 \cdot T(\frac{n}{2}) + \Theta(n)) & n \geq 2 \end{cases} \tag{27}$$

Possiamo risolverla utilizzando il metodo iterativo, sapendo che dovremo fare $i = \log_2 n$ iterazioni:

$$
\begin{aligned}
T(n) = 2 \cdot T(\frac{n}{2}) + c \cdot n = 2 \cdot (2 \cdot T(\frac{n}{4}) + c \cdot \frac{n}{2}) + c \cdot n = \\
= 4 \cdot T(\frac{n}{4}) + c \cdot n + c \cdot n = 4 \cdot (2 \cdot T(\frac{n}{8}) + c \cdot \frac{n}{4}) + 2 \cdot c \cdot n = \\
= 8 \cdot T(\frac{n}{8}) + c \cdot n + 2 \cdot c \cdot n = 8 \cdot T(\frac{n}{8}) + 3 \cdot c \cdot n = \\
= \ldots = 2^i \cdot T(\frac{n}{2^i}) + i \cdot c \cdot n = 2^{\log_2 n} \cdot T(1) + \log_2 n \cdot c \cdot n = \\
= n \cdot \Theta(1) + c \cdot n \cdot \log n = \Theta(n \cdot \log n)
\end{aligned} \tag{28}
$$

9.3.2 Albero di ricorsione

9.3.3 Master's Theorem

Quando si tratta di risolvere equazioni di ricorrenza **bilanciate**, è possibile utilizzare il Master's Theorem.

$$
T(n) = \begin{cases} \Theta(1) & n \leq k \\ a \cdot T(\frac{n}{b}) + f(n) & n > k \end{cases} \tag{29}
$$

L'intuizione consiste nel fare un confronto tra $f(n)$ e $n^{\log_b a}$, ovvero quante volte viene eseguito il passo ricorsivo.
Ci sono tre casi possibili:

- **Minore**: $f(n) = O(n^{\log_b a - \epsilon})$ per qualche costante $\epsilon > 0$.
  $f(n)$ cresce **polinomialmente** più lentamente di $n^{\log_b a}$ di un fattore n^ϵ.
  Soluzione: $T(n) = \Theta(n^{\log_b a})$

  Esempio 9.4. Data la seguente equazione di ricorrenza:

  $$T(n) = 9 \cdot T(\frac{n}{3}) + n \tag{30}$$

  Abbiamo che $a = 9$, $b = 3$, $f(n) = n$, $n^{\log_3 9} = n^2$. Possiamo dedurre quindi che, per un $\epsilon = 1$:

  $$f(n) = n = O(n^{\log_3 9 - \epsilon}) = O(n) \tag{31}$$

- **Uguale**: $f(n) = \Theta(n^{\log_b a} \cdot \ln^k n)$ per qualche costante $k \geq 0$.
  $f(n)$ e $n^{\log_b a}$ crescono allo stesso modo.
  Soluzione: $T(n) = \Theta(n^{\log_b a} \cdot \ln^{k+1} n)$

- **Maggiore**: $f(n) = \Omega(n^{\log_b a + \epsilon})$ per qualche costante $\epsilon > 0$.
  $f(n)$ cresce **polinomialmente** più in fretta di $n^{\log_b a}$ di un fattore N^ϵ e rispetta la **condizione di regolarità**: $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ per qualche costante $c < 1$ e $n > n_0$.
  Soluzione: $T(n) = \Theta(f(n))$

Osservazione 9.3.1. Il Master's Theorem si può usare solamente quando $f(x)$ cresce **polinomialmente** più in fretta o lentamente di $n^{\log_b a}$. Ad esempio se avessimo $f(x) = \log n$ non potremmo utilizzarlo.