Algoritmi e Strutture Dati

Luca Seggiani

11 maggio 2024

1 Introduzione agli algoritmi

Un'algoritmo non è altro che un procedimento che descrive una sequenza di passi ben definiti atti a risolvere un problema, o più schematicamente:

- Insieme finito di istruzioni
- Accetta un input e restituisce un output
- Ogni istruzione deve essere:
  - ben definita
  - eseguibile in tempo finito da un'agente di calcolo
- Può eventualmente usufruire di una memoria per stadi intermedi.

noi tratteremo di algoritmi aritmetici. Alcuni esempi di tali algoritmi possono essere;

- L'algoritmo di Euclide per il calcolo del MCD di due numeri
- Il setaccio di Eratostene per il calcolo dei primi numeri primi

Si può dire che l'algoritmo è l'essenza computazionale di un dato programma, notando però che:

$$\text{Algoritmo} \neq \text{Programma}$$

in quanto l'algoritmo è slegato dal linguaggio di implementazione, la macchina e l'ambiente su cui gira, mentre il programma equivale al suo codice ormai completamente definito.

2 Esempi basilari di algoritmi

Si riportano due semplici algoritmi ed alcune loro caratteristiche e possibili ottimizzazioni.

Algoritmo di Euclide

Nella sua definizione più semplice, l'algoritmo di Euclide si basa sul teorema: *"Il MCD fra 2 numeri è il MCD del più piccolo fra i due e la loro differenza"* e si implementa, in pseudocodice C++, nella forma:

```
int gcd(int a, int b) {
  while(a != b) {
    if(a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

una chiara ottimizzazione possibile è il passaggio dall'operazione di sottrazione a quella di sottrazione ripetuta (divisione), nel seguente modo:

```
int betterGcd(int a, int b) {
  while(b > 0) {
    int temp = b;
    b = a % b;
    a = temp;
  }
  return a;
}
```

si nota che la complessità dell'algoritmo è di $O(n)$.

Setaccio di Eratostene

Il setaccio di Eratostene è un algoritmo che ci permette di trovare tutti i numeri primi fra un insieme dei primi n naturali. Un approccio naive potrebbe essere quello di dividere ogni numero per ogni suo predecessore, ottenendo una complessità nel caso peggiore di $O(n^2)$. Un'alternativa più efficiente di questo algoritmo potrebbe considerare una tabella con i primi n naturali, e successivamente cancellare ogni multiplo di ogni numero primo trovato finora. raggiunto un numero il cui quadrato è maggiore di n, l'algoritmo si arresta e i numeri rimasti sono i primi cercati. In pseudocodice:

```
void sieve(int n) {
  bool num[n];

  //preparazione array
  num[0] = false; num[1] = true;
  for(int i = 2; i < n; i++) {
    num[i] = true;
```

```
  }

  int i = 2;
  while(i*i < n) {
    while(num[i] == false) i++;

    int m = 2;
    while(i * m < n) {
      num[i * m] = false;
      m++;
    }
    i++;
  }

  for(int i = 0; i < n; i++) {
    if(num[i] == true) {
      cout << i << " ";
    }
  }
}
```

che ottiene una complessità temporale di $O(n \log \log n)$.

3 Nozioni sugli algoritmi

Definito un determinato problema, si può utilizzare un'algoritmo su una sua certa istanza, analizzando quindi le seguenti caratteristiche:

- Problema: il problema da risolvere (e.g. trovare i primi n primi)
- Dimensione dell'istanza: la dimensione della mole di dati su cui dovrò lavorare nell'istanza attuale (e.g. il valore di n)
- Modello di calcolo: un modello che associa ad ogni operazione effettuata dall'algoritmo un certo costo per il mio ambiente
- Correttezza: se il mio algoritmo ottiene effettivamente la risposta giusta!

oltre alle caratteristiche riportate, è poi fondamentale la valutazione della

4 Complessità

La complessità di un'algoritmo è una funzione che associa alla dimensione del problema il costo della sua risoluzione, sia esso in termini di memoria

o tempo (in questo caso, *complessità temporale*). La dimensione dell'istanza dipende dai suoi stessi dati, ed è in generale opportuno determinare, su una istanza generica:

- Caso peggiore (worst-case): scenario dove l'algoritmo ottiene il costo di esecuzione maggiore
- Caso migliore (best-case): scenario dove l'algoritmo ottiene il costo di esecuzione minore
- Caso medio (average-case): scenario medio di esecuzione dell'algoritmo, necessita di un'analisi statistica dei dati delle istanze in entrata.

Si ricorda che l'efficienza di un'algoritmo dipende solo dal modello di calcolo adottato, ed è slegata dall'efficienza di un qualsiasi ambiente di esecuzione.

Posso ad esempio definire, per un determinato algoritmo P, la funzione complessità:

$$T_P(n)$$

che determina la complessità temporale di P al variare della dimensione d'istanza n. Prendiamo in esempio un semplice algoritmo che cerca il massimo di un vettore:

```
int max(int vett[], int n) {
  int m = a[0];
  for(int i = 0; i < n; i++) {
    if(m > a[i]) m = a[i];
  }
  return m;
}
```

possiamo adesso definire un modello di calcolo molto semplice, che associa il costo di 1 ad ogni operazione di assegnamento o confronto. Tenendo conto dell'assegnamento iniziale, quello dell'istruzione di ritorno, e gli assegnamenti e confronti eseguiti per l'inizializzazione e l'esecuzione del ciclo for, otteniamo:

$$1 + 1 + 4(n - 1) + 1 + 1 = 4n$$

ovvero la nostra funzione complessità dell'algoritmo. Per poter trattare di complessità senza però perdersi nelle specifiche, visto che l'unico fattore di interesse è la scalabilità dell'algoritmo, occorre definire la

5 Notazione O grande

Definiamo la notazione $O(f(n))$ nel seguente modo:

$$f(n) \in O(g(n)) \Leftrightarrow \exists n_0, \quad c > 0 \quad \text{t.c.} \quad \forall n > n_0 : f(n) \leq cg(n)$$

si definisce poi una serie di regole per l'utilizzo di tale notazione:

- fattori costanti: ammesso una certa costante k, si ha:

$$O(f(n)) = O(kf(n))$$

- somma:

$$f(n) \in O(g(n)) \Rightarrow f(n) + g(n) \in O(g(n))$$

- prodotto:

$$f(n) \in O(f_1(n)), \quad g(n) \in O(g_1(n)) \Rightarrow f(n)g(n) \in O(f_1(n)g_1(n))$$

- transitività:

$$f(n) \in O(g(n)) \wedge g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$$

- costanti:

$$\forall k, \quad k \in O(1)$$

- potenze:

$$m \leq p, \quad n^m \in O(n^p)$$

- polinomi:

$$p(x) \in R[x], \quad deg(p) = m, \quad p(x) \in O(n^m)$$

si nota che esistono funzioni tra di loro incommensurabili. Inoltre, per qualsiasi k:

$$\forall k, \quad n^k \in O(a^n), \quad \forall n > 1$$

ovvero qualsiasi polinomiale è sempre migliore di ogni esponenziale.

Per continuare la discussione della complessità degli algoritmi, occorre poi definire anche la:

Notazione Ω grande
La notazione Ω grande definisce effettivamente l'opposto della O grande:

$$f(n) \in \Omega(g(n)) \Leftrightarrow \exists n_0, \quad c > 0 \quad \text{t.c.} \forall n > n_0 : f(n) \geq cg(n)$$

Notazione Θ grande
La notazione Θ grande correla due funzioni che condividono lo stesso ordine di complessità:

$$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$$

oppure ancora:

$$f(n) \in \Theta(g(n)) \Leftrightarrow \exists n_0 \quad \exists c_1, c_2 \quad \text{t.c.} \quad \forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)$$

per la notazione Ω e Θ grande esistono inoltre le regole:

- antisimmetria O grande / Ω grande:

$$f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$$

- simmetria Θ grande:

$$f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$$

- per Θ grande e Ω grande valgono le regole definite per O grande, ovvero dei fattori costanti, somma e prodotto, transtività, potenze e polinomi. Si nota soratutto che per ogni polinomio:

$$p(x) \in R[x], \quad deg(p) = m, \quad p(x) \in \Theta(n)$$

in una nota un'attimo più formale, si può affermare che le notazioni O grande, Ω grande e Θ grande stabiliscono sull'insieme delle funzioni naturali $f(n)$ delle classi di equivalenza, rispettando effettivamente le tre proprietà di riflessività, simmetria e transitività. Inoltre, si può dire che l'appartenenza ad una classe di complessità stabilisce fra funzioni una relazione d'ordinamento, nell'ordine:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^m) < O(m^n) < O(n^n)$$

Si ricorda inoltre che queste notazioni considerano solamente l'andamento *asintotico* di una funzione, in quanto tutto ciò che accade prima di n_0 è effettivamente irrilevante all'analisi che vogliamo fare (almeno dal punto di vista della scalabilità).

6 Complessità degli algoritmi iterativi

Si presenta ora un semplice modello di calcolo per programmi scritti in pseudolinguaggio C++, compatibile con i tre paradigmi della programmazione strutturata (esecuzione sequenziale, condizionale ed iterativa).

Quello che vogliamo definire è una funzione del tipo:

$$\text{C[costrutti del linguaggio]} \rightarrow \text{classi di complessità}$$

useremo inoltre le abbreviazioni:

$$\text{V: costanti, I: variabili, E: espressioni, C: comandi}$$

iniziamo con semplici operazioni di assegnamento e lettura variabili, valutazioni di espressioni, selezione da vettori nonché l'istruzione di salto *return*:

$$C[V] = C[I] = O(1)$$

$$C[E_1 \circ E_2] = C[E_1] + C[E_2]$$

$$C[I[\,]] = C[E]$$

$$C[I = E] = C[E]$$

$$C[I[E_1] = E_2] = C[E_1] + C[E_2]$$

$$C[\text{return } E] = C[E]$$

si definisce poi per l'esecuzione sequenziale:

$$C[C_1, ..., C_n] = C[C_1] + ... + C[C_n]$$

per l'esecuzione condizionale:

$$C[\text{if}(E)C] = C[E] + C[C]$$

$$C[\text{if}(E)\, C_1 \text{ else } C_2] = C[E] + \max(C[C_1], C[C_2])$$

e per l'esecuzione iterativa ($O(g(n))$ è il numero di iterazioni in funzione della dimensione d'istanza):

$$C[\text{for}(E_1, E_2, E_3)C] = C[E_1] + C[E_2] + (C[C] + C[E_2] + C[E_3]) \cdot O(g(n))$$

$$C[\text{while}(E)C] = C[E] + (C[C] + C[E]) \cdot O(g(n))$$

si definisce inoltre per la chiamata di funzione:

$$C[F(E_1, ..., E_n)] = C[E_1] + C[E_n] + C[C...C]$$

7 Caratterizzazione di caso peggiore, migliore e medio

Definiamo matematicamente le nozioni di caso peggiore, migliore e medio sull'insieme di istanze In:

- Caso peggiore:
$$T_{worst} = max_{In}(\text{tempo}(I))$$
- Caso migliore:
$$T_{best} = min_{In}(\text{tempo}(I))$$
- Caso medio (la funzione $P(I)$ restituisce la probabilita di verificarsi di una certa istanza):
$$T_{average} = \sum_{In}(\text{tempo}(I) \cdot P(I)), \quad P(I) \in [0, 1]$$
da cui possiamo ovviamente dire:
$$T_{worst} \leq T_{average} \leq T_{best}$$

8 Complessità di algoritmi di uso comune

Iniziamo la discussione di alcuni algoritmi di ordinamento di vettori in loco (in-place), basati sul confronto, ammettendo l'accesso diretto in memoria in tempo $O(1)$. Definiamo innanzitutto una funzione di scambio con ausilio di variabile temporanea:

```
void swap(int& a, int& b) {
  int temp = a;
  a = b;
  b = temp;
}
```

Selection sort

Il selection sort lavora in maniera incrementale su n elementi, selezionando iterativamente l'i-esimo elemento e confrontandolo con glin-i elementi successivi in modo da scambiarlo con il minimo trovato. Dopo n-1 iterazioni il vettore risultaordinato. In pseudocodice:

```
void selectionSort(int vett[], int n) {
  for(int i = 0; i < n - 1; i++) {
    int m = i;
```

```
    for(int j = i + 1; j < n; j++) {
      if(vett[j] < vett[m]) m = j;
    }
    scambia[vett[m], vett[i]];
  }
}
```

il selection sort presenta caratteristiche identiche nel suo caso migliore e peggiore, effettuando sempre le stesse operazioni su qualsiasi tipo di istanza. In particolare, la complessità dipende dal numero di confronti:

$$(n-1)+(n-2)+(n-3)+...+2+1=\sum_{i=1}^{n} i=\frac{n(n-1)}{2}\in O(n^2)$$

e il numero di scambi, che è $O(n)$. Alternativa al selection sort può essere il

Bubble sort

Il bubble sort (algoritmo di ordinamento a bolle) non adotta un'approccio incrementale, ma si limita a scambiare da destra verso sinistra le coppie adiacenti non ordinate n-1 volte, finchè il vettore non risulta ordinato. In pseudocodice:

```
void bubbleSort(int vett[], int n) {
  for(int i = 0; i < n - 1; i++) {
    for(int j = n-1; j > i; j--) {
      if(vett[j] < vett[j - 1]) scambia(vett[j], vett[j - 1])
   ;
    }
  }
}
```

a differenza del selection sort, il bubble sort un numero sia di confronti che di scambi pari a $O(n^2)$. Questo apparente difetto è però compensato da una possibile ottimizzazione disponibile in fase di scorrimento del vettore: basterà infatti controllare eventuali scorrimenti senza scambi per poter ottenere una complessità temporale in caso migliore di $O(n)$ (caso di scorrimento singolo al primo ciclo). In pseudocodice:

```
void betterBubbleSort(int vett[], int n) {
  for(int i = 0; i < n - 1; i++) {
    bool q = false;
    for(int j = n-1; j > i; j--) {
      if(vett[j] < vett[j - 1]) {
        scambia(vett[j], vett[j - 1]);
        q = true;
      }
    }
    if(q == false) {
```

```
      break;
    }
  }
}
```

si può in generale applicare i limiti asintotici sia al costo di un singolo algoritmo che alla complessità di un intero programma. Possiamo ad esempio definire, su un qualsiasi problema:

- Limite asintotico superiore: il minimo ordine di complessità superiore a tutti gli ordini di complessità degli algoritmi atti a risovlere il problema (in sostanza caso peggiore)
- Limite asintotico inferiore il massimo ordine di complessita inferiore a tutti gli ordini di complessità degli algoritmi atti a risolvere il problema, quindi il miglior caso possibile per un qualsiasi algoritmo.

  Ad esempio, per quanto riguarda l'ordinamento di vettori monodimensionali, il limite asintotico inferiore è pari a $O(n \log n)$.

Ricerca lineare L'algoritmo di ricerca lineare di una certa chiave x su un certo vettore scansiona ogni singolo elemento confrontandolo alla suddetta, ottenendo complessità (appunto) lineare $O(n)$. In pseudocodice:

```
int linearSearch(int vett[], int n, int x) {
  int b = 0;
  for(int i = 1; i < n; i++) {
    if(vett[i] == x) b = 1;
    return b;
  }
}
```

si nota che il caso migliore (quello in cui il primo elemento è quello cercato) ha complessità $O(1)$.

Divisioni ripetute

Si riporta un algoritmo che divide un certo numero per 2 ciclicamente fino ad azzerarlo (ovviamente sul gruppo degli interi), magari per volerlo convertire in base 2:

```
int div_2(int n) {
   int i = 0;
   while(n > i) {
   n = \frac{n}{2}; i++;
   }
   return i;
}
```

possiamo quindi estrapolare una misura della complessità sul caso peggiore (quello in cui n è potenza di 2):

$$n = 2^m \Leftrightarrow m = \log_2 n \rightarrow O(log(n))$$

Ricera binaria

L'algoritmo di ricerca binaria (o ricerca dicotomica) effettua la stessa operazione della ricerca lineare, ma parte dall'assunto che il vettore sia già stato ordinato per approfittare di una notevole ottimizzazione, che porta la complessità temporale a $O(log(n))$. L'ottimizzazione usata si basa sul confronto della chiave con l'elemento centrale del vettore (e non di ogni suo elemento), che permette di dimezzare ciclicamente il vettore ottenendo quindi una riduzione a velocità logaritmica della dimensione d'istanza. In pseudocodice l'implementazione ricorsiva:

```
int binarySearch(int vett, int min, int max, int x) {
  if(max < min) return -1;
  int med = (min + max) / 2;
  if(vett[med] == key) {
    return med;
  }
  if(vett[med] > key) {
    return binarySearch(0, med - 1, key, dest);
  } else {
    return binarySearch(med + 1, max, key, dest);
  }
}
```

Si nota inoltre l'implementazione (probabilmente più efficiente) iterativa:

```
int binSearch_it(int A[], int x, int l, int r) {
  int m = (1 + r) / 2;
  while(A[m] != x) {
    if(x < A[m])
      r = m - 1;
    else // x > A[m]
      l = m + 1;
    if(l > r)
      return -1;
    m = (1 + r) / 2;
  }
  return m;
}
```

analizzando formalmente la complessità, otteniamo che il numero di passi totale è:

$$n_{passi} = \frac{n}{2^i}, \quad O(log(n))$$

merita attenzione il fatto che l'ordinamento del vettore prima della ricerca può portare a complessità complessive (bel pasticcio) addirittura di $O(n^2)$, e si dovrebbe quindi valutare con attenzione le casistiche di utilizzo.

9 Introduzione ai programmi ricorsivi

Iniziamo con l'esempio classico del calcolo del fattoriale. Fissato $0! = 1$, la definizione iterativa è:

$$n! = 1 \times 2 \times 3 \times ... \times n \quad n > 0$$

un'altra possibilite definizione, forse più naturale, è quella ricorsiva:

$$n! = n * (n-1)!, \quad n > 0$$

Vediamo i due approcci in pseudocodice:

```
  int fact(int n) {
    if(n == 0)
      return 1;
  int a = 1;
  for(int i = 1; i <= n; i++)
    a = a*i;
  return a;
}
```

```
int factRic(int n) {
  if(n == 0)
    return 1;
  return n * factRic(n-1);
}
```

possiamo dare definizioni ricorsive anche di algoritmi più semplici, come ad esempio la moltiplicazione:

```
int mult(int x, int y) {
int(x == 0)
  return 0;
  return y + mult(x - 1, y);
}
```

proviamo a fare l'unrolling ("srotolamento") di questo algoritmo: poniamo una condizione di partenza mult(3,4).

```
mult(3,4)
  4 + mult(2, 4)
    4 + mult(1, 4)
      4 + mult(0, 4) -- fine ricorsione
-> 3 * 4 = 12
```

prendiamo l'algoritmo per il calcolo di un numero pari:

```
int pari(int x) {
  if(x == 0) return 1;
  if(x == 1) return 2;
  return pari(x-2);
}
```

oppure l'algoritmo di Euclide nella sua formulazione ricorsiva:

```
int MCD(int x, int y) {
  if(x == y) return x;
  if(x < y) return MCD(x, y - x);
  return MCD(x - y, y);
}
```

chiararamente, tutti gli esempi precedentemente riportati rispettano una serie di regole:

- Regola 1: individuare il caso base in cui la funzione è immediatamente definita
- Regola 2: effettuare le chiamate ricorsive su un sottinsieme strettamente minore dei dati
- Regola 3: assicurarsi che la serie di chiamate ricorsive decade sempre su un caso base (in modo da terminare la ricorsione, che in caso contrario proseguirebbe all'infinito).

vediamo un'esempio di funzioni errate, cioè che non rispettano queste regole:

```
int pari_errata(int x) {
  if(x == 0) return 1;
  //fa che non sia dispari! regole 1 e 2 violate!
  return pari_errata(x - 2);
}
```

```
int MCD_errata(int x, int y) {
  if(x == y) return x;
  if(x < y) return MCD_errata(x, y-x);
  return MCD_errata(x, y);
  //nessuna restrizione del dominio: regola 3 violata!
}
```

10 Programmi ricorsivi su liste

Diamo una definizione di lista:

- Una sequenza vuota è una lista
- un elemento seguito da una lista è una lista

diamo una definizione della struttura Elem*, nodo della mia lista, in pseudocodice C++:

```
struct Elem {
  int inf;
  Elem* next;
}
```

vediamo un algoritmo per calcolare la lunghezza di una lista:

```
int length(Elem* p) {
  if(p == NULL) return 0;
  return 1 + length(p->next);
}
```

oppure un'algoritmo che restituisce il numero di volte che un nodo x compare in una lista:

```
int howMany(Elem* p, int x) {
  if(p == NULL) return 0;
  return (p->inf == x) + howMany(p->next, x);
}
```

altri algoritmi definiti sulle liste possono essere:

```
int belongs(Elem* l, int x) {
  if(l == NULL) return 0;
  if(l->inf == x) return 1;
  return belongs(l->next, x);
}
```

```
void tailDelete(Elem*& l) { //passaggio per riferimento
  if(l == NULL) return;
  if(l-> next == NULL) {
    delete l;
    l = NULL;
  }
  else tailDelete(l->next);
}
```

```
void tailInsert(Elem*& l, int x) {
  if(l == NULL) {
    l = new Elem;
```

```
    l->inf = x;
    l->next = NULL;
  }
  else tailInsert(l->next, x);
}
```

```
void append_1(Elem*& l1, Elem* l2) {
  if(l1 == NULL) l1 = l2;
  else append(l1->next, l2);
}
```

```
Elem* append_2(Elem* l1, Elem* l2) {
  if(l1 == NULL)
    return l2;
  l1->next = append(l1->next, l2);
  return l1;
}
```

11 Induzione

Riportiamo brevemente la definizione di induzione sui numeri naturali:

> Sia una certa proprietà P vera per un naturale n_0. Se è vero che $P(n) \Rightarrow P(n+1)$, allora P è vera per qualsiasi naturale $n > n_0$.

L'induzione può essere usata ad esempio per dimostrare che la somma dei primi n naturali è:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

introduciamo anche l'induzione completa:

> Se P vale per ogni naturale $m <= n$, e per ogni n vale per $n+1$, allora P vale per ogni naturale

e l'induzione ben fondata:

> Sia S un certo insieme ben ordinato. Se la proprietà P vale per i minimali di S, e per ogni elemento $E \in S$ P è vera su E se è vera su ogni elemento minore di E, allora P vale su tutto l'insieme S.
> (si noti che il totale abuso di notazione $P(x)$ indica P vera su x).

12 Complessità dei programmi ricorsivi

Facciamo il quadro della situazione, applicando quello che abbiamo visto finora e quello che avevamo già definito sugli algoritmi iterativi. Consideriamo il seguente algoritmo per il calcolo di $n!$:

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

l'algoritmo non fa altro che definire la relazione di ricorrenza:

$$T(0) = a, \quad T(n) = b + T(n-1)$$

andando a svolgere la serie di operazioni definite dall'algoritmo avremo:

$$T(0) = a$$

$$T(1) = b + a$$

$$T(2) = b + b + a = 2b + a$$

$$T(3) = b + 2b + a = 3b + a$$

$$\dots$$

$$T(n) = nb + a$$

l'ultima funzione, $T(n)$, non dipende da alcun passo ricorsivo e fornisce un'interpretazione della complessità dell'algoritmo (che è O(n)).

Selection sort

Applichiamo quanto visto finora ad un programma più complicato: il selection sort, formulato ricorsivamente.

```
void r_selectionSort(int* A, int m, int i = 0) {
  if(i == m - 1) return;
  int min = i;
  for(int j = i + 1; j < m; j++)
    if(A[j] < A[min]) min = j;
  exchange(A[i], A[min]);
  r_selectionSort(A, m, i + 1);
}
```

la relazione di ricorrenza in questo caso sarà:

$$T(1) = a, \quad T(n) = bn + T(n-1)$$

che potremo svolgere come:

$$T(1) = a$$

$$T(2) = 2b + a$$

$$T(3) = 3b + 2b + a$$

$$...$$

$$T(n) = (n + n - 1 + n - 2 + ... + 2)b + a = (\frac{n(n+1)}{2} - 1)b + a$$

13 Introduzione al quicksort

L'algoritmo quicksort è un'algoritmo di ordinamento particolarmente efficiente, schematicamente così definito:

- Scelta di un perno
- Divisione dell'array in 2 sottoinsiemi: elementi minori del perno ed elementi maggiori del perno
- Esegui la stessa operazione, ricorsivamente, sui 2 sottoinsiemi finchè non ne incontri uno di 2 elementi.

nel dettaglio, la divisione in due sottoinsiemi dell'array avviene mediante scambi fra elementi utilizzando due cursori (siano s e d), inizialmente posti, rispettivamente, ad inf e sup. Da qui in poi:

- Finché s $<$ d,
  porta avanti s finche A[s] è minore del perno;
  porta indietro d fino a che A[d] è maggiore del perno;
  scambia A[s] e A[d]; incrementa S e decrementa D.
- Se inf $<$ d ripeti il quicksort sulla parte da inf a d.
- Se s $<$ sup ripeti il quickosrt sulla parte da s a sup.

Vediamo infine lo pseudocodice:

```
1 void quickSort(int A[], int inf = 0, int sup = n -1) {
2   int perno = A[(inf + sup) / 2]; //prendo l'elemento
     centrale come perno
3   int s = inf, d = sup;
4   while(s < d) {
5     while(A[s] < perno) s++;
6     while{A[d] > perno} d--;
```

```
7       if(s > d) break;
8       exchange(A[s], A[d]);
9       s++;
10      d--;
11    }
12
13    if(inf < d)
14      quickSort(A, inf, d);
15    if(s < sup)
16      quickSort(A, s, sup);
17  }
```

14 Algoritmo Quicksort

Riprendiamo la trattazione dell'algoritmo quicksort (codice negli appunti presi il 5 marzo 2024). Possiamo innanzitutto dire che la complessità della fase non ricorsiva dell'algoritmo (scelta del perno e successiva divisione in parti del vettore) non richede mai più passaggi di quanti sono gli elementi del vettore ed è quindi di O(n). Dovrò però a questo punto considerare le due possibili chiamate ricorsive successive. Posso definire la seguente relazione di ricorrenza:

$$T(1) = a$$

$$T(n) = bn + T(k) + T(n-k)$$

notiamo come la posizione del perno k sia fondamentale all'efficienza dell'algoritmo. Nel caso di un perno estremamente sbilanciato, e.g. k = 1, avremo:

$$T(1) = a$$

$$T(n) = bn + T(n-1)$$

posto invece (come vorremmo) $k = \frac{n}{2}$, avremo:

$$T(1) = a$$

$$T(n) = bn + 2T(\frac{n}{2})$$

che diventerà:

$$T(n) = (n \log n)b + na$$

potremo a questo punto dire che la complessità dell'algoritmo e di $O(n^2)$ nel suo caso peggiore, ma di O(nlogn) nel suo caso medio, cosa che lo rende molto più efficiente di altri algoritmi di ordinamento sui vettori.

15 Torre di Hanoi

La formulazione del problema della torre di Hanoi prevede 3 paletti, con 3 dischetti di diametro decrescente, impilati sul primo paletto. A questo punto si chiede di portare i 3 dischetti, nello stesso ordine, sull'ultimo paletto, con la limitazione fondamentale che non è possibile sposta più di un cerchio alla volta, né di mettere un cerchio più grande su uno più piccolo. Chiamati i 3 paletti A, B e C, definiamo una funzione "trasferisci" che sposta n cerchi dal paletto A al paletto C:

trasferisci una torre di n cerchi da A a C
se n = 1
sposta il cerchio da A a C
altrimenti
trasferisci la torre degli n - 1 cerchi più piccoli da A a B usando C come paletto ausiliario
sposta il cerchio più grande da A a C
trasferisci la torre degli n-1 cerchi più piccoli da B a C usando A come paletto ausiliario

in codice, sfruttando la ricorsione:

```
1  void hanoi(int n, pal A, pal B, pal C) {
2    if(n == 1)
3      sposta(A, C);
4    else {
5      hanoi(n - 1, A, C, B);
6      sposta(A, C);
7      hanoi(n-1, B, A, C);
8    }
9  }
```

da cui la relazione di ricorrenza:

$$T(1) = a$$

$$T(n) = b + 2T(n-1)$$

che potremo sviluppare in:

$$T(n) = (2^{(n-1)} - 1)b + 2^{(n-1)}a$$

ovvero un T(n) di $O(2^n)$!

16 Altri algoritmi ricorsivi

Ricerca lineare ricorsiva

Poniamo un algoritmo ricorsivo che implementi la ricerca lineare:

```
int RlinearSearch (int A[], int x, int m, int i = 0) {
  if(i == m) return 0;
  if(A[i] == x) return 1;
  return RlinearSearch(A, x, m, i+1);
}
```

dove la chiamata ricorsiva è fatta su una dimensione di istanza di n - 1. La relazione per ricorrenza di questo algoritmo si risolve e decade in una complessità di O(n).

Ricerca binara ricorsiva

Ripetiamo quanto fatto prima sull'algoritmo di ricerca binaria:

```
int binSearch(int A[], int x, int i = 0, int j = m - 1) {
  if(i > j) return 0;
  int k = (i + j) / 2;
  if (x == A[k]) return 1;
  if (x < A[k])
    return binSearch(A, x, i, k - 1);
  else
    return binSearech(A, x, k + 1, j);
}
```

notiamo che le ultime due chiamate ricorsive, simili a quelle del quicksort, sono in questo caso mutualmente esclusive, e la complessità dell'algoritmo decade quindi in $O(\log n)$. Ricordiamo che la ricerca binaria vale solo su vettori ordinati.

Ricerca ricorsiva

Esiste un algoritmo simile alla ricerca binaria ma applicabile a vettori non ordinati:

```
int Search(int A[], int x, int i = 0, int j = n - 1) {
  if(i > j) return 0;
  int k = (i + j) / 2;
  if(x == A[k])
    return 1;
  return Search(A, x, i, k - 1) || Search(A, x, k + 1, j);
}
```

la complessità dell'algoritmo, data la relazione di ricorrenza:

$$T(0) = a$$

$$T(n) = b + 2T(n/2)$$

sarà identica alla sua controparte iterativa, ovvero di O(n).

17 Classificazione di alcuni relazioni di ricorrenza

Diamo adesso una classificazione di alcune delle relazioni di ricorrenza che potremmo incontrare. La maggior parte degli algoritmi che risultano in tali relazioni sono i cosiddetti "divide et impera", ovvero dividi e comanda, cieè algoritmi che dividono l'istanza del problema su cui operano in sottoproblemi più semplici da risolvere separatamente.

```
void divideEtImpera(S) {
  if(|S| <= m)
    //risolvi direttamente il problema
  else
  //dividi in due istanze separate
    divideEtImpera(S_1)
    ...
    divideEtImpera(S_2)
    //combina le due istanze
}
```

classifichiamo quindi alcune delle relazioni di ricorrenza che potremo ottenere da algoritmi simili:

$$T(0) = d$$
$$T(n) = c + T(\frac{n}{2})$$

Con complessità $O(\log n)$,

$$T(0) = d$$
$$T(n) = c + \frac{2T}{n/2}$$

Con complessità $O(n)$,

$$T(0) = d$$
$$T(n) = cn + 2T(\frac{n}{2})$$

Con complessità $O(n \log n)$.
In generale, se abbiamo:

$$T(n) = d, \quad n = 1$$
$$T(n) = c + aT(\frac{n}{b}), \quad n > 1$$

possiamo allora dire che:

$$T(n) \in O(\log n), \quad a = 1$$

$$T(n) \in O(n^{\log_b a})$$

e se abbiamo:

$$T(n) = d, \quad n <= m$$

$$T(n) = hn^k + aT\frac{n}{b}, \quad n > m$$

diremo:

$$T(n) \in O(n^k), \quad a < b^k$$

$$T(n) \in O(n^k \log n), \quad a = b^k$$

$$T(n) \in O(n^{log_b a}), \quad a > b^k$$

Ulteriori generalizzazioni si possono trovare nell'enunciato del Master Theorem, trovato ad esempio nel volume del Cormen.

18 Introduzione agli algoritmi di teoria dei numeri

Prendiamo adesso agloritmi la cui complessità è calcolata sulla base del numero di cifre che compongono il numero stesso. Ad esempio:

- La somma ha complessità lineare
- La moltiplicazione ha complessità quadratica

Moltiplicazione veloce fra interi non negativi
Notiamo la seguente proprietà dei naturali:

$$A = A_s 10^{\frac{n}{2}} + A_d$$

dove A_s e A_d sono due parti del numero A diviso (rispetto alle cifre) a metà.

19 Insertion sort

L'insertion sort utilizza il metodo più intuitivo per ordinare un vettore: si prende ad ogni iterazione il primo elemento, si confronta con ogni suo precedente finchè non si trova la sua posizione corretta, e poi si spostano tutti i precedenti necessari di una posizione in avanti per lasciargli posso. Notiamo che nell'implementazione in pseudocodice il while usato per spostare gli elementi ha la conseguenza di lasciare il primo elemento con un valore duplicato: usiamo allora la variabile ausiliaria current per rimettere il valore giusto al suo posto.

```
1  void sort(int* vett, int dim) {
2    int current = 0;
3    int p = 0;
4    for(int i = 1; i < dim; i++) {
5      current = vett[i];
6      p = i - 1;
7
8      while(p >= 0 && vett[p] > current) {
9        vett[p + 1] = vett[p];
10       p--;
11     }
12     vett[p + 1] = current;
13   }
14 }
```

Possiamo fare l'analisi della complessità e trovare che, come per tutti gli algoritmi di ordinamento in-place, la complessità massima è di $O(n^2)$. Il worst case sarà chiaramente quello di un vettore ordinato al contrario, cioe in ordine decrescente.

20 Debugging

Il processo di debugging consiste nel rimuovere eventuali errori dal codice scritto. Si possono distinguere le seguenti categorie di debugging:

- Visuale: osservando lo stato del programma in un qualsiasi momento (ad esempio scrivendo sul buffer di uscita le informazioni che ci interessano);
- Debugger: utilizzando appositi software chiamati debugger (GDB, DDD) che forniscono alcuni strumenti utili al debugging quali i breakpoint, l'analisi della memoria in tempo reale, ecc...;

- Compilatore: utilizzando i messaggi di errore forniti dallo stesso compilatore;
- Analisi della memoria: utilizzando software come Valgrind per gestire correttamente la memoria ed individuare leak e segmentazioni.

21 Programmi in memoria dinamica

Utilizziamo la programmazione in memoria dinamica nel caso in cui le dimensioni d'istanza del problema che ci interessa non siano note a tempo di compilazione. Diciamo ad esempio di voler immagazinare una serie di n numeri. Potremmo voler usare una lista, ma a quel punto dovremmo accettare di dover richiedere tutte le nostre letture sulla struttura dati in tempo n. Utilizzando invece un vettore dinamico, avremmo una complessità di $O(1)$ per ogni accesso. Tutte queste strutture dati possono essere trovate nell'implementazione della libreria standard, ovvero la:

Standard Template Library (STL)
La libreria STL definisce una serie di container, ovvero strutture dati predefinite che possono venire utilizzate nel nostro codice. Vediamo per esempio un vettore definito attraverso la STL:

```
vector<int> stlArray
```

qui vector indica il tipo di container, lal voce <int> definisce il tipo di dati immagazinati nel container, e stlArray il nome del vettore. Per aggiungere un elemento al vettore, potremo ad'esempio;

```
stlArray.push_back(val);
```

chiamando la funzione membro push_back sulla stlArray (altro non è che un'istanza di classe) con argomento val. Per leggere un valore dal vettore potremmo usare la ridefinizione dell'operatore []:

```
stlArray[index];
```

potremo ottenere iteratori sulla testa e sulla coda del vettore con:

```
stlArray.begin();
stlArray.end();
```

ed ottenerne la dimensione con:

```
stlArray.size();
```

Riguardo agli algoritmi di ordinamento visti finora, la libreria STL fornisce una funzione:

```
sort(stlArray.begin(), stlArray.end());
```

dove i due argomenti sono iteratori generici tra i quali verrà effettuato l'ordinamento. L'algoritmo dell'ordinamento è implementation-dependant, cioò dipende dall'implementazione usata, ma assicura sempre una complessità di $O(n \log n)$.

Ulteriori informazioni su qualsiasi componente della STL si può trovare su siti come:
`https://cplusplus.com/reference`

22 Algoritmi di teoria dei numeri

Studiamo adesso una categoria di algoritmi la cui complessità è calcolata prendendo come misura il numero di cifre che compongono il numero. Ad esempio:

- L'addizione ha complessità $O(n)$;
- La moltiplicazione ha complessità $O(n^2)$.

Vediamo ad esempio:

Moltiplicazione veloce fra interi non negativi
Ogni numero A di n cifre può essere visto come:

$$A = A_s 10^{\frac{n}{2}} + A_d$$

A questo punto il prodotto AB sarà:

$$AB = A_s B_s 10^{\frac{n}{2}} + A_d B_d$$

che può essere riscritto come:

$$AB = A_s B_S 10^n + ((A_s + A_d)(B_s + B_d) - A_s B_s - A_d B_d) 10^{\frac{n}{2}} + A_d B_d$$

dove compaiono tre moltiplicazioni fra interi non con n ma con $\frac{n}{2}$ cifre. In codice, questo si tradurrà in:

```
int mult(int A, int B, int n) {
  if(n = 1) return A * B;
  else {
    As = parte sinistra di A; Ad = parte destra di A;
    Bs = parte sinistra di B, Bd = parte destra di B;
    int x_1 = As + Ad; int x_2 = Bs + Bd;
    int y_1 = mult(x_1, x_2, n / 2);
    int y_2 = mult(A, B, n / 2);
```

```
      int y_3 = mult(Ad, Bd, n / 2);
      int s_1 = left_shift(y_2, n);
      int s_2 = left_shift(t_1 - t_2- t_3, n / 2);
      return s_1 + s_2 + y_3;
    }
  }
```

Dove l'operazione left_shift(h, n) scorre h di n posti a sinstra, introducento n 0 ($\times 10^n$). Abbiamo tre chiamate ricorsive alla funzione mult, ciascuna su una dimensione di istanza di $\frac{n}{2}$. Da qui la relazione di istanza:

$$T(1) = d, \quad T(n) = hn + 3T(\frac{n}{2})$$

Da cui otteniamo che $T \in O(n^{\log_2 3}) = O(n^{1.59})$.

Relazioni lineari

Relazioni lineari in forma:

$$T(0) = d, \quad T(n) = bn^k + a_1 T(n-1) + a_2 T(n-2) + a_r T(n-r)$$

sono polinomiali soltanto se esiste al più una sola chiamata ricorsiva ($a_i = 1$ e $a_j = 0$ con $j \neq i$). Applichiamo questo teorema al problema del calcolo della serie di Fibonacci. Ricordiamo la classica formulazione ricorsiva:

$$f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2} \quad (0, 1, 3, 5, 8, 13...)$$

da cui si ricava:

$$\phi = \lim_{n \to +\infty} \frac{f_n}{f_{n-1}} = 1.618...$$

nota come sezione aurea (o rapporto aureo). Cerchiamo allora di implementare un algoritmo ricorsivo che restituisca n cifre della serie di Fibonacci, ad esempio come:

```
int fibonacci(int n) {
  if(n < 2) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

La relazione di ricorrenza di questo algoritmo è:

$$T(0) = T(1) = d, \quad T(n) = b + T(n-1) + T(n-2)$$

da cui chiaramente avremo complessità esponenziale (chiaramente poco). Cerchiamo allora di definire un algoritmo iterativo:

```
int fibonacci(int n) {
  int k; int j = 0; int f = 1;
  for(int i = 1; i <= n; ++i) {
    k = j; j = f; f = k + j;
  }
  return j;
}
```

visto che il nostro numero di passaggi $g(n)$ è n, avremo una complessità totale di $O(n)$. Questo però significa soltanto che la nostra implementazione ricorsiva di partenza era inefficiente. Troviamone un altra:

```
int fibonacci(int n; int a = 0; int b = 1) {
  if(n == 0)  return a;
  return fibonacci(n - 1, b, a + b);
}
```

adesso abbiamo una singola chaiamata ricorsiva su una dimensione di istanza di $(n-1)$, da cui ricaviamo complessità $O(n)$.

23 Mergesort

Il mergesort è un algoritmo di ordinamento basato sull'unione, cioè sulla suddivisione del vettore in sottoinsiemi di dimensione minore, l'ordinamento di quei sottoinsiemi e la loro successiva riunificazione in un unico vettore. In pseudocodice:

```
void mergeSort(sequenza S_1) {
if(S.size() <= 1) return;
else {
  //dividi S in 2 sottosequenze S_1 e S_2 di uguale lunghezza
  sequenza S_2;
  split(S_1, S_2);

  mergeSort(S_1);
  mergeSort(s_2);

  //fondi S_1 e S_2
  merge(S_1, S_2);
  }
}
```

proviamo un'implementazione basata sulle liste:

```
void mergeSort(elem*& s_1) {
  if(s_1 == NULL || s_1->next == NULL) return;
  elem* s_2 = NULL;
  split(s_1, s_2);
  mergeSort(s_1);
```

```
  mergeSort(s_2);
  merge(s_1, s_2);
}
```

sarà necessario, dalla ricorrenza:

$$T(n) = bn + 2T(\frac{n}{2})$$

che una singola chiamata ricorsiva non sia peggiore di $O(n \log n)$. Vediamo le implementazioni della split e della merge:

```
void split(elem*& s_1, elem*& s_2) {
  if(s_1 == NULL || s_1->next = NULL) return;
  elem* p = s_1->next;
  s_1->next = p->next;
  p->next = s_2;
  s_2 = p;
  split(s_1->next, s_2);
}
```

La funzione split() divide la lista in due "sottoliste", estraendo alternativamente un elemento (da mettere in una nuova lista) e lasciandone un altro nella lista di partenza.

```
void merge(elem*$ s_1, elem*& s_2) {
  if(s_2 == NULL) return;
  if(s_1 == NULL) {
    s_1 = s_2;
    return;
  }
  if(s_1->inf <= s_2->inf)
    merge(s_1->next, s_2);
  else {
    merge(s_2->next, s_1);
    s_1 = s_2;
  }
}
```

La funzione merge() invece scorre le due liste contemporaneamente confrontando di volta in volta i due elementi concorrenti in testa ad entrambe e scegliendo il minore.

24 Mergesort su vettori non in-place

Implementiamo un'algoritmo di ordinamento mergesort sui vettori non in-place, ovvero che sfrutta un vettore ausiliario implementato come vettore STL (per l'ausilio della funzione push_back). Il corpo del mergesort sarà:

```
void mergeSort(int arr[], int beg, int end) {
    if(beg + 1 < end) {
        int mid = (beg + end) / 2;
        mergeSort(arr, beg, mid);
        mergeSort(arr, mid, end);
        merge(arr, beg, mid, end);
    }
}
```

dove la chiamata ricorsiva è fatta sulle due sottoarray di dimensione dimezzata dell'array di partenza. Notare che la suddivisione si ferma giunti ad array di dimensione monadica. La merge è implementata come segue:

```
void merge(int arr[], int beg, int mid, int end) {
    int iS = beg;
    int iD = mid;

    vector<int> temp;

    while(true) {
        if(arr[iS] <= arr[iD]) {
            temp.push_back(arr[iS++]);
            if(iS >= mid) {
                while(iD < end) temp.push_back(arr[iD++]);
                break;
            }
        } else {
            temp.push_back(arr[iD++]);
            if(iD >= end) {
                while(iS < mid) temp.push_back(arr[iS++]);
                break;
            }
        }
    }

    for(int i = 0; i < temp.size(); i++) {
        arr[i + beg] = temp[i];
    }

}
```

Si inizializzano due indici, iS e iD, che partono rispettivamente dall'inizio e dal punto medio della sottoarray. Da qui in poi si scorrono entrambi gli indici, selezionando ogni volta l'elemento minore, finchè uno dei due non sfora il suo bound (il punto medio per iS, la fine della sottoarray per iD). A questo punto si effettua l'unrolling completo della parte di sottoarray rimanente attraverso l'indice rimasto libero. Infine si copiano i valori selezionati e ordinati, che sono stati scritti su un vettore dinamico, sul vettore di partenza a partire dalla

posizione d'inizio della sottoarray. Una tipica esecuzione di questo algoritmo, su dimensione di istanza $n = 15$, sarà:

```
111 86  23  33  21  21  95  92  15  4 49  65  43  9 15
11  23  86  33  21  21  95  92  15  4 49  65  43  9 15
11  23  86  33  21  21  95  92  15  4 49  65  43  9 15
11  23  86  21  33  21  95  92  15  4 49  65  43  9 15
11  23  86  21  33  21  95  92  15  4 49  65  43  9 15
11  23  86  21  21  33  95  92  15  4 49  65  43  9 15
11  21  21  23  33  86  95  92  15  4 49  65  43  9 15
11  21  21  23  33  86  95  15  92  4 49  65  43  9 15
11  21  21  23  33  86  95  15  92  4 49  65  43  9 15
11  21  21  23  33  86  95  4 15  49  92  65  43  9 15
11  21  21  23  33  86  95  4 15  49  92  43  65  9 15
11  21  21  23  33  86  95  4 15  49  92  43  65  9 15
11  21  21  23  33  86  95  4 15  49  92  9 15  43  65
11  21  21  23  33  86  95  4 9 15  15  43  49  65  92
4 9 11  15  15  21  21  23  33  43  49  65  86  92  95
4 9 11  15  15  21  21  23  33  43  49  65  86  92  95
11  23  86  33  21  21  95  92  15  4 49  65  43  9 15
11  23  86  33  21  21  95  92  15  4 49  65  43  9 15
11  23  86  21  33  21  95  92  15  4 49  65  43  9 15
11  23  86  21  33  21  95  92  15  4 49  65  43  9 15
11  23  86  21  21  33  95  92  15  4 49  65  43  9 15
11  21  21  23  33  86  95  92  15  4 49  65  43  9 15
11  21  21  23  33  86  95  15  92  4 49  65  43  9 15
11  21  21  23  33  86  95  15  92  4 49  65  43  9 15
11  21  21  23  33  86  95  4 15  49  92  65  43  9 15
11  21  21  23  33  86  95  4 15  49  92  43  65  9 15
11  21  21  23  33  86  95  4 15  49  92  43  65  9 15
11  21  21  23  33  86  95  4 15  49  92  9 15  43  65
11  21  21  23  33  86  95  4 9 15  15  43  49  65  92
4 9 11  15  15  21  21  23  33  43  49  65  86  92  95
4 9 11  15  15  21  21  23  33  43  49  65  86  92  95
```

25 Ordinamenti multivalore

Supponiamo di avere vettori formati da elementi con più di un valore su cui è stabilita una relazione d'ordine, come ad esempio:

```
  struct Richiesta {
    int id_;
    int prio_;
    Richiesta(int id, int prio) : id_(id), prio_(prio) {}
  }
```

nella STL, potremo usare la funzione sort sfruttando l'argomento comparatore:

```
sort(first, last comparatore);
```

definiamo ad esempio una funzione di confronto:

```
bool confrontaRichiesta(Richiesta r_1, Richiesta r_2) {
  if (r_1.id_ < r_2.id_) retirm true;
  else if {r_1.id_ == r_2.id_} {
    if(r_1.prio_ > r_2.prio_) return true;
    else return false;
  } else return false;
}
```

potremo adesso passare come argomento la funzione confrontaRichiesta(Richiesta, Richiesta), alla funzione sort() su un qualsiasi vettore di elementi Richiesta, così che venga usata nei confronti dell'ordinamento.

26 Debugging della memoria con Valgrind

Valgrind è un applicativo divenuto praticamente standard per la programmazione in c++, che controlla la corretta gestione della memoria dinamica. Per fare il debugging di un programma attraverso Valgrind, occorrerà prima di tutto compilare il nostro eseguibile abilitando i flag di debugging:

```
g++ -g programma.cpp -o programma
```

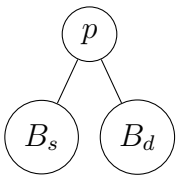
e chiamare poi Valgrind sull'eseguibile creato:

```
valgrind programma
```

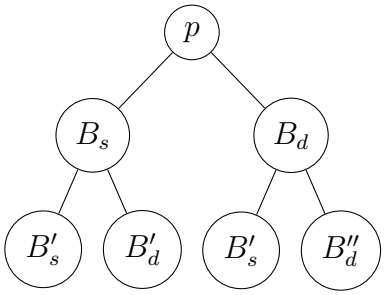
27 Alberi binari

Un'albero binario è definito come segue:

- L'albero vuoto è un'albero binario.
- Un nodo p più due alberi binari B_s e B_d formano un'albero binario.



a loro volta B_s e B_d saranno alberi binari, quindi:



...

a questo punto diremo che:

- p è la **radice** dell'albero;
- B_s è il **sottoalbero sinistro** di p;
- B_d è il **sottoalbero destro** di p;
- gli alberi sono etichettati.

diciamo inoltre che ogni albero senza sottoalberi (o figli) è una foglia. Un nodo sarà padre rispetto al loro figlio, i nodi precedenti saranno glli antecedenti e quelli successivi i discendenti. Il livello di un nodo è il numero dei suoi antecedenti, mentre il livello di un'albero è il livello massimo dei suoi nodi.

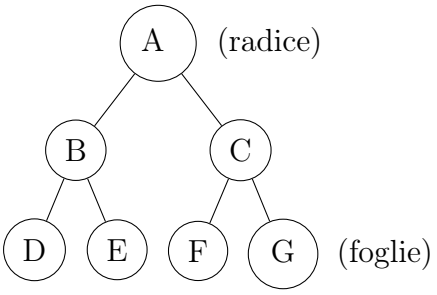
28 Alberi binari

Riprendiamo la trattazione delgli alberi binari. Avevamo detto che un albero è definito come segue:

- L'albero vuoto è un'albero binario.
- Un nodo p più due alberi binari B_s e B_d formano un'albero binario.

Ad esempio, assumiamo che il livello di un albero sia -1. Il livello della radice sarà 0, e il livello dell'albero totale sarà il più lungo cammino fra la radice e una foglia. Un albero binario si dice inoltre etichettatto quando ad ogni nodo è associato un nome, o etichetta.
Una possibile implementazione della struttura dati albero binario in pseudo-codice potrebbe essere:

```
struct Node {
  int inf;
  Node* sx;
  Node* dx;
};
```



Gli alberi binari si prestano all'implementazione, su di essi, di algoritmi ricorsivi. Avremo infatti:

- Caso base: albero vuoto
- Passo ricorsivo: radice e i due sottoalberi

Gli algoritmi più comuni su alberi binari sono quelli di: linearizzazione, ricerca, inserimento e cancellazione di nodi. Vediamo la prima categoria:

Linearizzazione di alberi
Vediamo come possiamo linearizzare una struttura dati ad albero. Una linearizzazione di un albero è una sequenza contenente i nomi dei suoi nodi. Le più comuni linearizzazioni, dette **visite**, sono:

- **Ordine anticipato** (*preorder*):
  Si stampa prima di tutto la radice, e poi si chiama ricorsivamente sui sottoalberi sinistri e destro, stampando quindi i nodi appena trovati e poi procedendo di livello nell'albero:

```
void preOrder(Node* albero) {
  if(albero == NULL) return;
  else {
    print(radice);
    preOrder(albero->sx);
    preOrder(albero->dx);
  }
}

```

- **Ordine differito** (*postorder*):
  Si scorre prima ricorsivamente tutto l'albero, raggiungendo l'ultimo nodo a sinistra e stampandolo. Le stampe si susseguono poi attraverso il meccanismo della ricorsione in coda:

```
void postOrder(Node* albero) {
  if(albero == NULL) return;
  else {
    preOrder(albero->sx);
    preOrder(albero->dx);
    print(radice);
  }
}

```

- **Ordine simmetrico** (*inorder*):
  Si stampa la radice dopo la stampa del sottoalbero sinistro e prima della stampa del sottoalbero destro, rispettando quindi quello che sarebbe l'"ordine" naturale dell'albero:

```
void postOrder(Node* albero) {
  if(albero == NULL) return;
  else {
    preOrder(albero->sx);
    print(radice);
    preOrder(albero->dx);
  }
}

```

Alberi binari bilanciati
Un albero binario si dice bilanciato quando ogni suo nodo (tranne quelli all'ultimo livello) ha due figli. Il numero di nodi di un albero binario bilanciato è $2^{(k+1)} - 1$, e il numero di foglie è 2^k.
Alberi binari quasi bilanciati
Un albero binario è quasi bilanciato quando tutti i livelli fino al penultimo sono bilanciati. Un albero bilanciato è anche quasi bilanciato.
Alberi pienamente binari Un albero sii dice pienamente binario quando tutti i nodi tranne le foglie hanno 2 figli. Il numero di nodi interni è uguale al numero di foglie meno 1.

Complessità della linearizzazione
Studiamo la complessità di una visita fatta su un albero. Abbiamo chiaramente due passi ricorsivi, più un operazione di confronto e di una di lettura che richiedono entrambe tempo $O(1)$. Il problema è posto dal fatto che

non conosciamo accuratamente la dimensione dei sottoproblemi al momento della chiamata ricorsivo. Il caso migliore sarebbe quello di sottoalberi bilanciati, ovvero con numero uguale di nodi figli. Possiamo in ogni caso porre la ricorrenza:

$$T(0) = 0, \quad T(n) = b + T(n_s) + T(n_d), \quad n_s + n_d = n - 1, \quad n > 0$$

Il caso particolare bilanciato è il seguente:

$$T(0) = a, \quad T(n) = b + 2T(\frac{n-1}{2})$$

con complessità in caso migliore di $T(n) \in O(n)$. Possiamo inoltre esprimere la complessità in funzione dei livelli, applicando quanto visto prima sugli alberi binari bilanciati:

$$T(0) = a, \quad T(k) = b + 2T(k-1)$$

dove abbiamo complessità $T(k) \in O(2^k)$. Chiaramente la complessità è maggiore, ma possiamo dire che in genere $k < n$, e anche di molto.

Altre funzioni su alberi binari

Poniamoci il problema di contare nodi e foglie di un albero binario. Una possibile soluzione è:

```
//conta nodi
int nodes(Node* tree) {
  if(tree == NULL) return 0;
  return 1 + nodes(tree->sx) + nodes(tree->dx);
}
//conta foglie
int leaves(Node* tree) {
  if(tree == NULL) return 0;
  if(!tree->sx && !tree->dx) return 1; //foglia
  return leaves(tree->sx) + leaves(tree->dx);
}
```

Entrambe le funzioni hanno formula di ricorrenza simile a quelle già viste sulle visite, e quindi complessità $O(n)$.

Ricerca di un'etichetta su un albero binario

Implementiamo una funzione che restituisce un puntatore al primo nodo con etichetta n che trova in visita anticipata. Se il nodo non viene trovato, la funzione restituisce NULL.

```
Node* findNode(int n, Node* tree) {
  if(tree == NULL) return NULL;
  if(tree->inf = n) return tree;
  Node* a = findNode(n, tree->left);
```

```
  if(a) return a;
  else return findNode(n, tree->right);
}
```

La complessità è sempre $O(n)$.

Cancellazione dell'albero binario

Vediamo la cancellazione dell'albero, che avremo per esempio bisogno di fare nel caso l'albero fosse allocato in memoria dinamica e si rendesse necessaria la sua deallocazione:

```
void delTree(Node* &tree) {
  if(tree != NULL) {
    delTree(tree->left);
    delTree(tree->right);
    delete tree;
    tree = NULL;
  }
}
```

Inserzione di un nodo su un albero binario

Vediamo una funzione che inserisce un nodo *child* come figlio di un nodo *parent*, sinistro se $c = l$ e destro se $c = r$. Se l'albero è vuoto, inserisce *child* come radice. Se *parent* non esiste o ha già il figlio richiesto, la funzione non modifica l'albero.

```
int insertNode(Node* &tree, int child, int parent, char c) {
  if(tree == NULL) { //albero vuoto
    tree = new Node;
    tree->label = child;
    tree->left = tree->right = NULL;
    return 1;
  }
  Node* a = findNode(parent, tree); //cerca parent
  if(parent == NULL) return 0;
  if(c == 'l' && !a->left) { //figlio sinistro
    a->left = new Node;
    a->left->inf = child;
    a->left->left = a->left->right = NULL;
    return 1;
  }
  if(c == 'r' && !a->right) { //figlio destro
    a->right = new Node;
    a->right->inf = child;
    a->right->left = a->right->right = NULL;
    return 1;
  }
}
```

29 Alberi generici

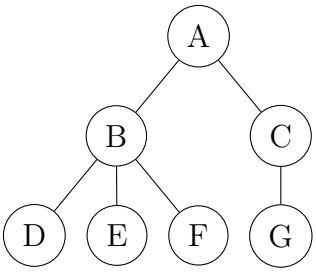
Generalizziamo la definizione di albero binario introducendo l'albero generico. Il concetto chiave sarà sempre quello di una radice, da cui partiranno però non uno ma una sequenza di sottoalberi. Formalmente:

- Un nodo è un albero
- Un nodo più una sequenza di alberi $A_1, ..., A_n$ è un albero.

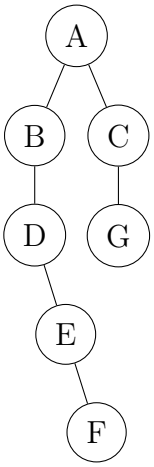
Notiamo come fra i sottoalberi di un nodo di albero generico non è stabilito alcun ordinamento, mentre nei sottoalberi di alberi binari c'era una chiara distinzione fra sottoalbero sinistro e sottoalbero destro.
La rappresentazione di una struttura dati di tipo albero generico potrebbe essere complicata: l'approccio naive è infatti quello di immagazzinare insieme ad ogni etichetta, una lista dei sottoalberi presenti. Questo è però particolarmente inefficiente, in quanto l'accesso ad ogni sottoalbero richederebbe una scansione lineare della lista dei sottoalberi. Un approccio migliore è quello figlio-fratello: a partire dal nodo radici, definiamo una struttura effettivamente analoga a quella di un albero binario, con la differenza, però, che il puntatore di sinistra indica adesso il prossimo nodo su un livello più basso (figlio), mentre quello di destra indica il prossimo nodo sullo stesso livello (fratello). L'albero binario così generato prende il nome di trasformato dell'albero generico. Una possibile implementazione della struttura dati albero generico sarà quindi:

```
struct Node {
  int inf;
  Node* child;
  Node* sibling;
};
```

che rappresenterà un'albero del tipo:



Attraverso il suo trasformato:



Molte delle funzioni definite su alberi binari valgono (con qualche modifiche) sugli alberi generici, ad esempio la visita anticipata:

```
void preOrder(albero) {
  print(radice);
  preOrder(albero->1);
  ...
  preOrder(albero->n);
}
```

Proprio per la proprietà riportata prima, invece, la visita simmetrica risulta impossbile: non si può stabilire un sottoalbero "centrale" di un nodo. Notiamo, che nella rappresentazione figlio-fratello, la visita anticipata del trasformato corrisponde alla visita trasformata dell'albero generico, e la visita simmetrica del trasformato corrisponde alla visita in differita dell'albero generico. Come per gli alberi binari, la complessità delle operazioni su alberi generici è di $O(n)$ sul numero di elementi.

Altre funzioni su alberi generici

Poniamoci nuovamente il problema di contare nodi e foglie di un albero, stavolta generico. La soluzione è del tutto analoga a quella dell'albero binario, con la sola differenza che un nodo è classificato come foglia quando il suo puntatore child è NULL, e ciò non significa che non possa avere altri sibling, anch'essi foglie.

```
//conta nodi
int nodes(Node* tree) {
  if(tree == NULL) return 0;
  return 1 + nodes(tree->child) + nodes(tree->sibling);
}
//conta foglie
int leaves(Node* tree) {
  if(tree == NULL) return 0;
  if(!tree->child) return 1 + leaves(tree->sibling); //foglia
```

```
  return leaves(tree->child) + leaves(tree->sibling);
}
```

Inserzione di un nodo su un albero generico
Riportiamo una funzione che inserisca un nodo come ultimo fratello di un certo sibling:

```
void addSibling(int sibling, Node* &list) {
  if(list == NULL) {
    list = new Node;
    list->inf = sibling;
    list->child = list->sibling = NULL;
  } else addChild(sibling, list->sibling)
}
```

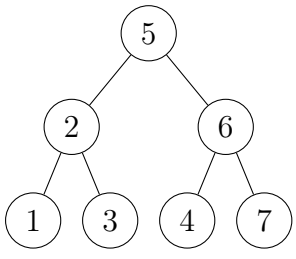
E infine altra che inserisca un nodo come ultimo figlio di un certo parent:

```
int insert(int child, int father, Node* &tree) {
  Node* a = findNode(father, tree);
  if(a == NULL) return 0;
  addChild(child, a->child);
  return 1;
}
```

notando che la funzione findNode(int, Node*) è analoga a quella definita sugli alberi binari.

30 Alberi Binari di Ricerca

Un'albero binario di ricerca è un albero binario tale che per ogni nodo p, i nodi del sottoalbero sinistro di p hanno etichetta minore dell'etichetta di p. Di contro, i nodi del sottoalbero destro avranno etichettta maggiore dell'etichetta di p. Si stabilisce un ordine tra i nodi, più forte del semplice meccanismo antecedente-precedente, in base all'ordinamento tra le etichette. Ad esempio:



Notiamo che non possono esistere nodi doppioni, e che l'ordinamento risulta infine da sinistra verso destra, ergo una stampa simmetrica dell'albero binario di ricerca fornisce sostanzialmente una lista dei nodi in ordine crescente.

Possiamo definire sugli alberi di ricerca alcune operazioni, fra cui la ricerca di un nodo (ottimizzato!), l'inserimento ordinato di un nodo, e la cancellazione di un nodo.

Ricerca su alberi di ricerca

Si spera che, se l'hanno chiamati alberi di ricerca, almeno quella ci si possa fà per benino. Effettivamente la ricerca su alberi di ricerca si può eseguire sempre come ricerca binaria, essendo gli stessi sempre ordinati per definizione. Possiamo quindi implementare, in pseudocodice, la funzione ricorsiva:

```
Node* findNode(int n, Node* tree) {
  if(tree == NULL) return 0;
  if(n == tree->label) return tree;
  if(n < tree->label) return findNode(n, tree->left);
  return findNode(n, tree->right);
}
```

Questo ci permetterà di dividere per 2 la dimensione di istanza ad ogni passaggio, ottenendo una complessità di $O(\log n)$. Chiaramente, questo vale solo nel caso migliore: nel caso medio, il valore verrà fortemente influenzato dal bilanciamento (o mancanza di tale) del nostro albero. E' fondamentale avere alberi bilanciati per ottimizzare le operazioni di ricerca e inserzione. Nel caso l'albero sia completamente sbilanciato, si dice che è degenere, e le operazioni di ricerca costeranno su di esso $O(n)$.

Inserzione in alberi di ricerca

Inserire nodi in alberi di ricerca significa fare l'inserimento ordinato, per mantenere appunto l'ordine dell'albero:

```
void insertNode(int n, Node *& tree) {
  if(tree == NULL) {
    tree = new Node;
    tree->label = n;
    tree->left = tree->right = NULL;
  }
  if(n < tree->label) insertNode(n, tree->left);
  if(n > tree->label) insertNode(n, tree->right);
}
```

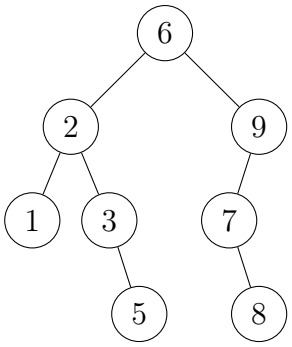
L'algoritmo aggiunge, se l'albero considerato è vuoto, l'elemento considerato. Sennò confronta l'elemento con i sottoalberi sinistri e destro dell'albero, e chiamandosi ricorsivamente sul ramo che rispetta l'ordinamento. La complessità è sempre $O(\log n)$, che può arrivare a $O(n)$ se l'albero è degenere.

Cancellazione di alberi di ricerca

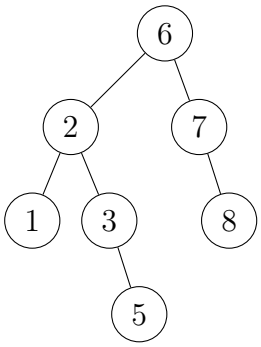
Poniamoci adesso il problema di cercare e successivamente cancellare un determinato nodo. Occorrerà innanzitutto trovare il nodo desiderato attraverso i meccanismi sopra definiti, e valutare poi le due possibili situazioni che potremmo incontrare:

- Se p (l'albero trovato) ha un sottoalbero vuoto, il padre di p viene connesso all'unico sottoalbero non vuoto di p.
- Se p ha entrambi i sottoalberi non vuoti si cerca il nodo con etichetta minore nel sottoalbero destro di p, si cancella e si mette la sua etichetta come etichetta di p.

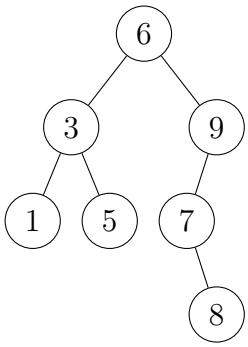
Visualizziamo i diversi casi:



rimuovere il 9 significa:



e rimuovere il 2 significa:



Ciò si può implementare con:

```
//helper (trova minimo)
void deleteMin(Node* &tree, int &m) {
  if(tree->left) deleteMin(tree->left, m);
  else {
    m = tree->label;
    Node* a = tree;
    tree = tree->right;
```

```
8       delete a;
9     }
10  }
11
12  //funzione
13  void deleteNode(int n, Node* &tree) {
14    if(tree) {
15      if(n < tree->label) {
16        deleteNode(n, tree->left);
17        return;
18      }
19      else if(n > tree->label) {
20        deleteNode(n, tree->right);
21        return;
22      }
23      else if(tree->left == NULL) {
24        Node* a = tree;
25        tree = tree->right;
26        delete a;
27        return;
28      }
29      else if(tree->right == NULL) {
30        Node* a = tree;
31        tree = tree->left;
32        delete a;
33        return;
34      }
35      else deleteMin(tree->right, tree->label);
36    }
37  }
```

L'helper deleteMin() trova il minimo, lo elimina e ne inserisce l'etichetta nell'integer m. Questo torna utile nel secondo caso, quando abbiamo intenzione di eliminare il nodo minimo e sostituirne l'etichetta al nodo eliminato. La funzione a questo punto si occupa soltanto di scorrere l'albero in maniera ricorsiva, applicando l'algoritmo giusto a seconda del caso incontrato. Nello specifico, finchè non trova il nodo desiderato prosegue di ricerca binaria. Quando raggiunge a una foglia che corrisponde, sceglie in base al numero di sottoalberi il percorso da prendere: nel caso esista solamente un sottoalbero, si elimina dopo aver sostituito il suo sottoalbero esistente a se stesso. Nel caso entrambi i sottoalberi esistano, chiama deleteMin() sul suo sottoalbero destro, passando la sua stessa etichetta per riferimento. Notiamo che il caso dove l'etichetta cercata non esiste è gestito dal controllo su tree in capo a entrambe le funzioni (albero vuoto $\rightarrow$ caso base).

Costo di riempimento di un'albero binario di ricerca

Vediamo quanto costa in funzione di n elementi da aggiungere, riempire un

albero binario di ricerca. Potremmo bovinamente dire $O(n \log n)$, visto che svolgo effettivamente n operazioni di costo $O(\log n)$. Il problema sta nel fatto che il costo di $O(\log n)$ vale nel caso ottimale, non nel caso generale: fosse considerato il caso peggiore, avremmo n volte qualcosa che ha complessita $O(n)$, che però verrebbe ogni volta chiamato su dimensioni di istanza $n-1$, da cui $O(n\frac{n-1}{2})$, ovvero $O(n^2)$. Dobbiamo allora ricordare l'albero di ricerca binaria è una struttura di dati estremamente efficiente, ma che la sua creazione ha il costo non indifferente di $O(n^2)$.

31 Alberi binari di ricerca, implementazione

Vediamo l'implementazione effettiva di un albero binario di ricerca, effettuata in C++ attraverso il meccanismo delle classi:

```
struct Node {
  int value;
  Node* left;
  Node* right;

  Node(int val):
    value(val), left(NULL), right(NULL) {}
};

class BinTree{
  Node root_;

public:
  BinTree() { root_ = NULL; }
  Node* getRoot() { return root_; }
};
```

La insert:

```
void insert(int val) {
  Node* node = new Node(val);
  Node* pre = NULL;
  Node* post = root_;

  while(post != NULL) {
    pre = post;
    if(val <= post->value)
      post = post->left;
    else
      post = post->right;
  }

  if(pre == nullptr) {
```

```
    root_ = node;
    return;
  }

  if(val <= pre->val)
    pre->right = val;
  else
    pre->left = val;

}
```

Funzioni per la ricerca di minimi e massimi:

```
Node* min() {
  Node* temp = root_;
  while(temp->left != NULL)
    temp = temp->left;
  return temp;
}

Node* max() {
  Node* temp = root_;
  while(temp->right != NULL)
    temp = temp->right;
  return temp;
}
```

Funzioni per la ricerca di la profondità di minimi e massimi:

```
int height(Node* tree) {
  int hLeft;
  int hRight;
  if(tree == NULL)
    return 0;
  hLeft = height(tree->left);
  hRight = height(tree->right);

  return(1 + max(hLeft, hRight));
}
```

Funzioni di ricerca di nodi:

```
Node* search(int val) {
  Node* temp = root_;
  while(temp != nullptr) {
    if(val == temp->value)
      return temp;
    if(val <= temp->value)
      search(temp->left);
    else
      search(temp->right);
  }
```

```
  return nullptr;
}
```

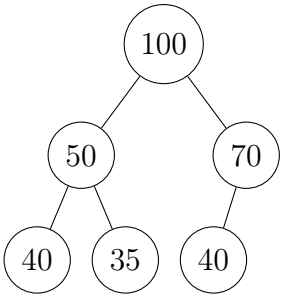
32 Stringhe STL

La libreria STL definisce nell'header <string> il tipo di dato stringa. Il tipo di dato stringa estende in qualche modo la stringa del comune C, ovvero un'array di tipi carattere terminata da un carattere specifico. Attraverso l'incapsulamento di questo tipo di dati, si possono avere diverse comodità, come ad esempio il conto della lunghezza della lunghezza della stringa senza necessità di scorrimenti, o funzioni membro come:

```
string stringa = "Il contrario di comunismo";
string stringa_2 = " e' egoismo";
string stringa_completa = stringa + stringa_2;
  //stringa_completa = "Il contrario di comunismo e' egoismo"
stringa.find("comunismo");
  //puntatore alla prima occorrenza di "comunismo", senno' :
    npos
stringa_2.compare("egoismo");
...
```

33 Struttura dati Heap

Un heap è un albero binario quasi bilanciato con la proprietà di avere i nodi dell'ultimo livello addossati a sinistra, e sopratutto che in ogni sottoalbero l'etichetta della radice è maggiore o uguale a quella di tutti i discendenti (la cosiddetta *heap property*). Vediamo un'esempio:



La visita in ordine anticipato di tale albero restituisce l'array:

$$100, 50, 70, 40, 35, 40$$

Il primo elemento dell'array sarà sempre la radice dell'heap. Si nota inoltre che il figlio sinistro di un qualsiasi elemento di indice i si trova con $2i+1$, il figlio destro con $2i+2$, e infine il padre con $\frac{i-1}{2}$.
Sugli heap sono definite operazioni di inserimento e estrazione dell'elemento maggiore. L'estrazione prevede complessità di $O(\log n)$ nel caso medio, e $O(n)$ nel caso peggiore (albero linearizzato). Vediamo intanto una possibile implementazione:

```
class Heap {
  int* h;
  int last; //indice dell'ultimo elemento
  void up(int);
  void down(int);
  void exchange(int i, int j) {
    int k = h[i]; h[i] = h[j]; h[j] = k;
  }
public:
  Heap(int);
  ~Heap(int);
  void insert(int);
  int extract();
}
```

Vediamo nello specifico costruttore e distruttore.

```
Heap::Heap(int n) {
  h = new int[n];
  last = -1;
}
Heap::~Heap() {
  delete[] h;
}
```

è chiaro come la struttura dati viene realizzata effettivamente con un vettore di interi, vettore che rappresenta la vista in ordine anticipato dell'albero che forma l'heap.

Inserimento in Heap

La strategia adottata per l'inserimento è quella di memorizzare l'elemento nella prima posizione libera dell'array, e far poi risalire l'elemento tramite scambi figlio padri finché l'heap property non è rispettata:

```
void Heap::insert(int x) {
  h[++last] = x;
  up(last);
}
```

Notiamo come viene sfruttata la funzione up() definita precedentemente per far risalire l'elemento. Vediamone l'implementazione:

```
void Heap::up(int i) {
  if(i > 0) {
    if(h[i] > h[(i - 2) / 2] {
      exchange(i, (i - 2) / 2);
      up((i - 2) / 2);
    }
  }
}
```

dove si sfrutta la proprieta per cui il padre di i è sempre $\frac{i-1}{2}$ per assicurare l'heap property.

Estrazione da Heap

La strategia adottata per l'estrazione è quella di restituire il primo elemento dell'array, mettere l'ultimo elemento al suo posto (e decrementare il valore last), e a questo punto far scendere l'elemento tramite scambi padre figlio per mantenere l'heap property. In codice:

```
int Heap::extract() {
  int r = h[0];
  h[0] = h[last--];
  down(0);
  return r;
}
```

Vediamo l'implementazione della down:

```
void Heap::down(int i) {
  int child = 2 * i + 1;
  if(child == last) {
    if(h(son) > h(i))
      exchange(i, last);
  } else if(child < last) {
    if(h[child] < h[child + 1])
      child++;
    if(h(child) > h(i)) {
      exchange(i, child);
      down(child);
    }
  }
}
```

Applicazioni dell'Heap

La struttura dati heap torna particolarmente utile nella realizzazione del tipo di dato astratto **coda con proprità**. Essa si tratta di una coda in cui gli elementi contengono, oltre all'informazione, un'intero che ne definisce la priorità. In caso di estrazioni l'elemento da estrarre dovrà essere ovviamente quello con maggiore priorità.

34 Heap Sort

Il meccanismo dell'heap permette di realizzare algoritmi di ordinamento. Basterà infatti creare un heap a partire da un certo vettore, estrarre il primo elemento (che andremo a disporre in fondo all'heap) e ripetere finché il vettore non è completamente ordinato. L'ordinamento risulterà come conseguenza dell'operazione ripetuta di estrazione, che ricordiamo comporta la preservazione dell'*heap property* attraverso scambi successivi. Più schematicamente, si trasforma un'array in un heap, e si segue n volte l'estrazione scambiando ogni volta il primo elemento dell'array con quello puntato da last. In codice:

```
void heapSort(int* vett, int n) { //n = dimensione dell'array
  buildHeap(vett, n);
  int i = n - 1;
  while(i > 0) {
    extract(vett, i);
  }
}
```

Vediamo quindi la funzione buildHeap() usata per costruire l'heap. Vorremo semplicemente eseguire la funzione down() sulla prima metà degli elementi dell'array, eseguendo quindi down() partendo dalla metà, e scorrendo fino al primo elemento.

```
void buildHeap(int* vett, int n) {
  for(int i = (n / 2) - 1; i >= 0; i--) {
    down(A,i, n);
  }
}
```

Questo richiederà una rielaborazione sia della down che della extract, in modo che possano lavorare su vettori arbitrari. (L'implementazione fornita insieme agli appunti si basa di per sé su un vettore STL, che può semplicemente fungere da contenitore per il vettore che vorremmo ordinare...) La complessità dell'inserimento a questo punto sarà di $O(n)$, mentre la complessità delle estrazioni ripetute darà una complessità finale di $O(n \log n)$, che non è affatto male.

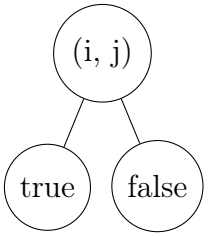
35 Limiti inferiori

Un problema si dice di ordine $\Omega(f(n))$ se non è possibile trovare un'algoritmo che lo risolva con complessità minore di $f(n)$. Tutti gli algoritmi che lo risolvono saranno quindi $O(n)$. Cercheremo adesso di individuare un limi-

te inferiore per la complessità dell'algoritmo, ovvero un limite massimo di ottimizzazione possibile. Per fare ciò avremo bisogno di un albero decisione.

Alberi di decisione

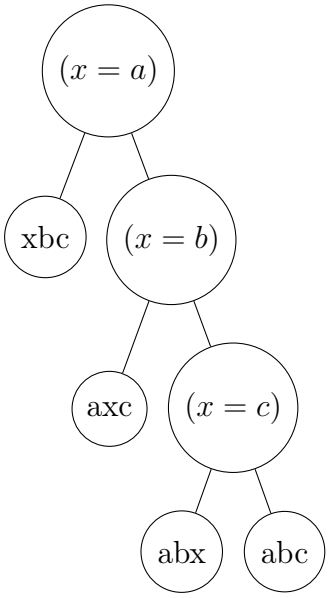
Un'albero di decisione si applica solamente agli algoritmi basati sui confronti, e che hanno complessità proporzionale al numero di confronti effettuatu duranti l'esecuzione dell'algoritmo. L'unità fondamentale di un albero di decisione è un confronto:



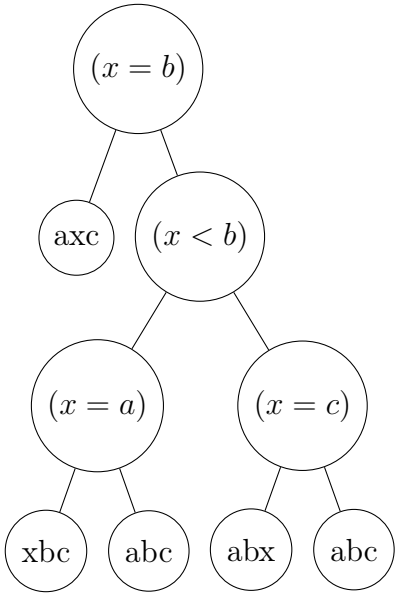
che può produrre una di due possibilità, vero o falso. A questo punto, possiamo dire che un'albero di decisione rappresenta un albero binario che corrisponde all'algoritmo che vogliamo analizzare. Per la precisione:

- Ogni **foglia** corrisponde ad un possibile esito dell'algoritmo;
- Ogni **cammino** lungo l'albero corrisponde ad una possibile esecuzione dell'algoritmo che restituisce un certo risultato, ovvero una certa foglia.

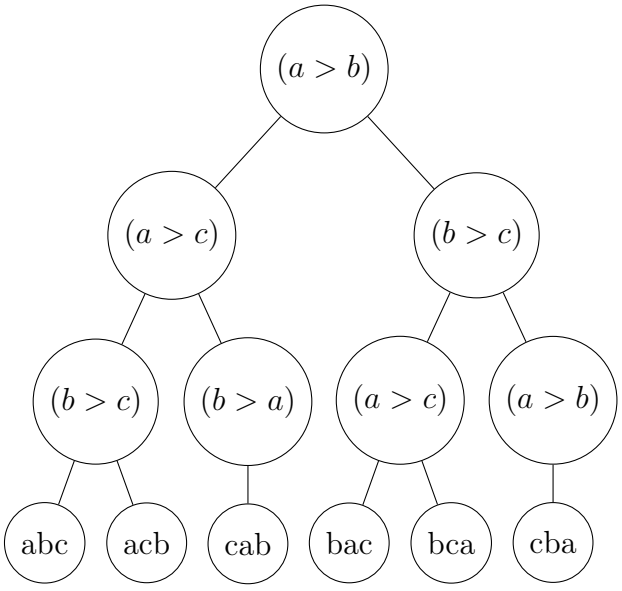
Ad esempio, vediamo l'albero di decisione della ricerca lineare, immaginando di cercare valori su stringhe in forma "abc":



Come vediamo, per dimensioni di stringa uguali a 3 avremo un albero di profondità 3, dove ogni elemento viene scansionato almeno una volta. Vediamo allora la ricerca binaria:



Possiamo infine vedere il selection sort su tre elementi:



Dove notiamo inoltre il confronto $a > b$ potrà essere eseguito due volte lungo un solito cammino. Questo sarà conseguenza naturale dell'esecuzione del selection sort.
Esiste quindi un'equivalenza fra alberi di decisione ed algoritmi. Chiedersi

qual'è l'algoritmo più efficiente per risolvere un dato problema significa chiedersi quale sia la lunghezza massima dei percorsi sull'albero binario corrispondente all'algoritmo. Sarà questa lunghezza a minimizzare la complessità dell'algoritmo (fornire un limite inferiore) nel suo caso peggiore. Si possono fare considerazioni simili per quanto riguarda il caso medio: basterà prendere la lunghezza media dei cammini su uno o più alberi di decisione. In questo caso però andranno fatte anche considerazioni statistiche sulla natura degli input forniti al nostro algoritmo, come va comunque fatto sempre quando si studia la complessità media.

Applicazioni degli alberi di decisione

Un'albero binario di k livelli ha al massimo 2^k foglie nel caso sia bilanciato. Possiamo vedere questa relazione dal senso opposto: un albero binario con s foglie ha almeno $\log_2(s)$ livelli. Gli alberi binari bilanciati minimizzano quindi sia il caso peggiore che quello medio: hanno $\log(s(n))$ livelli.

Vediamo il caso specifico dell'algoritmo di ordinamento: il numero di soluzioni è effettivamente il numero di permutazioni di un insieme di dimensione n, che si calcola come $n!$. Se il numero di soluzioni e $n!$, il cammino medio e massimo saranno $\log(n!) = n \log n$ (dalla formula di Sterling). Per la precisione, avremo che:

$$n! \sim \sqrt{2\pi x} \left(\frac{x}{e}\right)^x \in O(n \log n)$$

Questo ci suggerisce che il mergesort è ottimo, e il quicksort è ottimo nel caso medio. Per quanto riguarda le ricerche, $\log(n)$ è il caso migliore, ma non è sempre raggiungibile: si ha con la ricerca binaria. Esistono poi algoritmi che, per quanto possa sembrare assurdo, adottano alcune soluzioni per avere complessità minore del limite inferiore.

Counting sort

Diciamo di avere una restrizione sui nostri input: abbiamo n elementi, tutti compresi fra 0 e k. Il counting sort sfrutta queste prerogative per ottenere una complessità minore al limite inferiore. Chiaramente, il counting sort conviene solamente quando conosciamo i valori minimo e massimo degli elementi da ordinare. Lo svolgimento è questo:

- Per ogni valore dell'array, si contano i duplicati utilizzando un array ausiliario di dimensione k.
- Successivamente si ordinano i vaori tenendo conto dell'array ausiliario.

In sostanza si tiene conto solamente del numero di occorrenze di ogni valore in k, che vengono poi restituiti in ordine, contati una volta per ogni loro presenza nell'array di partenza. In codice, questo si traduce come:

```c
void counting_sort(int A[], int k, int n) {
  int i, j; int C[k + 1];
  for(i = 0; i <= k; i++) C[i] = 0;
  for(j = 0; j < n; j++) C[A[j]]++;
  j = 0;
  for(i = 0; i <= k; i++) {
    while(C[i] > 0) {
      A[j] = i;
      C[i]--;
      j++;
    }
  }
}
```

Notiamo che il counting sort è estraneo al meccanismo degli alberi di decisione, in quanto non è nemmeno basato sui confronti. Questo gli permette di "eludere" il limite inferiore. Per la precisione, ha complessità $O(n + k)$, e conviene quando k è $O(n)$. Inoltre, il counting sort necessita, a differenza degli algoritmi di ordinamento *in-place*, di memoria ausiliara.

Radix sort

Quando si conosce la lunghezza massima d dei numeri da ordinare, può essere utile implementare il radix sort. Il suo funzionamento è il seguente:

- Si eseguono d passate ripartendo, in base alla d-esima cifra, i numeri in k contenitori, dove k sarà il numero di possibili valori di una cifra.
- Si rilegge il risultato in un determinato ordine.

Il radix sort si basa sulla rappresentazione dell'elemento da ordinare per ottenere un'ordinamento in tempo inferiore a $n \log n$. Si nota che il numero k rappresenta tutti i possibili valori di una cifra, rappresentato il numero in base k. Le passate vengono solitamente svolte da destra verso sinistra, ovvero dalla cifra meno significativa alla più significativa (LSD (*"Least Significant Digit"*) vs MSD (*"Most Significant Digit"*)): questo è utile alla stragrande maggioranza delle applicazioni del radix sort (ordinamento di interi e ordinamento lessicografico).
Vediamo ad esempio come ordinare i numeri 190, 051, 052, 207, 088, 010 con un radix sort (d = 3, k = 10): innanzitutto si inseriscono i numeri nei contenitori in base al valore nell'ultima cifra:

```
010|   |   |   |   |   |   |   |   |
190|051|   |   |054|   |   |207|088|
---|---|---|---|---|---|---|---|---|---
0   |1  |2  |3  |4  |5  |6  |7  |8  |9
```

Si rileggono poi i numeri da sinistra verso destra, e dal basso verso l'alto: 190, 010, 051, 054, 207, 088. Si ripete quindi la stessa operazione sulla penultima cifra:

```
   |   |   |   |   |054|   |   |   |
207|010|   |   |   |051|   |   |088|190
---|---|---|---|---|---|---|---|---|---
0  |1  |2  |3  |4  |5  |6  |7  |8  |9
```

Si rileggono in modo analogo a prima: 207, 010, 051, 054, 088, 190. Si ripete per un'ultima volta l'operazione sulla prima cifra:

```
088|   |   |   |   |   |   |   |   |
054|   |   |   |   |   |   |   |   |
051|   |   |   |   |   |   |   |   |
010|190|207|   |   |   |   |   |   |
---|---|---|---|---|---|---|---|---|---
0  |1  |2  |3  |4  |5  |6  |7  |8  |9
```

Si leggono a questo punti i valori in modo analogo a prima, ottenendo il vettore ordinato: 010, 051, 054, 088, 190, 207.
Il radix sort, come il counting sort, non è basato sui confronti. La sua complessità è $O(d(n+k))$ dove d è la lunghezza delle sequenze e k è il numero possibile di valori di ogni cifra (quindi nell'esempio precedente $O(3(n+10))$). Necessita, come il counting sort, di memoria ausiliaria. E' particolarmente conveniente quando d è molto minore di n. Si può usare agevolmente per ordinare in ordine alfabetico sequenze di caratteri. Vediamo ad esempio uno spezzone di codice che ordina un vettore STL di stringhe di lunghezza len, con una chiave k che sarà uguale al numero di lettere nell'alfabeto anglosassone (26).

```
void radix(vector<string>& vec, int len) {
  vector<vector<string>> temp;
  temp.reserve(k);
  for(int t = 1; t <= len; t++) {
    for(int i = 0; i < vec.size(); i++) {
      temp[vec[i][len - t]-'@'].push_back(vec[i]);
    }
    int count = 0;
    for(int i = 0; i < k; i++) {
      for(int j = 0; j < temp[i].size(); j++) {
        vec[count] = temp[i][j];
        count++;
      }
      temp[i].clear();
    }
  }
}
```

Bucket sort

Tutti questi algoritmi sono in qualche modo varianti del bucket sort. L'algoritmo bucket sort si limita a scandire la lista, noto la sua variazione di elementi, ed a suddividere i suoi elementi in k "secchielli" in base al loro valore. I secchiellli vengono poi ordinati uno per uno, e infine si ricompone la lista completa ormai ordinata. Il radix sort è in questo modo una sorta di bucket sort ripetuto, senza alcun passo di ordinamento ma eseguita su diverse cifre degli elementi da ordinare. In pseudocodice, il bucketsort ha il seguente aspetto:

```
//helper
int getMax(int A[], int n) {
  int max = A[0];
  for(int i = 1; i < n; i++) {
    if(A[i] > max) max = A[i];
  }
  return max;
}

//funzione
void bucketSort(int A[], int n, int k) {
  int[][] buckets = new int[][k];
  int max = 1 + getMax(A, n);
  for(int i = 0; i < n; i++) {
    buckets[floor(k * A[i] / max)].push_back(A[i]);
  }
  for(int i = 0; i < k; i++) {
    buckets[i].sort;
  }
  int count = 0;
  for(int i = 0; i < k; i++) {
    for(int j = 0; j < buckets[i].size(); j++) {
      A[count] = buckets[i][j];
      count++;
    }
  }
}
```

36 Hashing

Abbiamo già avuto a che fare con algoritmi di ricerca: i nostri approcci finora si limitavano a scorrimenti completi della lista (ricerca lineare) o ad approcci divide et impera che si basavano sul preordinamento (ricerca binaria). Presentiamo adesso un metodo più (sotto determinate condizioni) efficiente per fare ricerche su array, non basato sui confronti.

Funzione di hashing

Il concetto alla base dell'hashing è la cosiddetta funzione di hashing. La funzione di hashing stabilisce una corrispondenza quanto più iniettiva e surgettiva fra il tipo di dati su cui vogliamo effettuare ricerche, e un intero i che sarà indice di un'array. Se ammettiamo di allocare un'array di dimensione k, il suo risultato sarà un intero modulo k, ovvero un valore compreso fra $0 < n < k - 1$. L'implementazione della funzione di hashing consiste spesso in qualche "riarrangiamento" del valore iniziale (il verbo *to hash* in inglese significa spezzettare, confondere...), previa elevamenti a potenze particolari o altre operazioni molto divergenti. Un'esempio di algoritmo di ricerca effettuata con questo metodo potrebbe essere:

```
bool hashSearch(T* A, T x) {
  int i = h(x);
  if(A[i] == x) return true;
  else return false;
}
```

La funzione ci restituisce effettivamente se l'elemento x è stato immagazzinato nell'array attraverso la funzione di hashing h(x).

Indirizzamento aperto

Un'approccio usato spesso per implementare la funzione di hashing è quella di definire una funzione modulo la dimensione dell'array k:

$$h(x) = f(x) \ \% \ k$$

Notiamo che questo significa rilasciare l'iniettività della funzione: potranno esistere più di valori in entrata che corrisponderanno alla stessa chiave generata da h(x). Cercare di immagazzinare entrambi questi valori risulterà in una cosiddetta collisione.

Gestione delle collisioni

Come abbiamo visto, la funzione h(x) non potrà mai essere veramente iniettiva: scelto un k necessariamente finito, se il nostro range di dati in entrata è illimitato (o comunque più grande di k) sarà impossibile stabilire una funzione di hashing inettiva. Si renderà necessario allora un meccanismo di difesa contro le eventuali (e inevitabili) collisioni. Un'approccio usato per l'eliminazione delle collisioni è quello della scansione lineare: se non si trova l'elemento al primo posto restituito dalla funzione di hashing, si scorre l'array progressivamente, finchè non si riesce a trovare l'elemento o si torna sull'elemento di partenza. Una possibile implementazione di questo algoritmo potrà essere:

```
bool hashSearch(T* A, int k, T x) {
  int i = h(x);
  for(int j = 0; j < k; j++) {
```

```
    int pos = (i + j) % k;
    if(A[pos] == -1) return false;
    if(A[pos] == x) return true;
  }
  return false;
}
```

A questo punto potremo inserire con:

```
int hashInsert(T* A, int n, T x) {
  int i = h(x);
  for(int j = 0; j < n; j++) {
    int pos = (i + j) % n;
    if(A[pos] == -1) {
      A[pos] = x;
      return 1;
    }
  }
  return 0;
}
```

Scelta del k

Poniamo di voler immagazinare un'insieme di dati di dimensione minima n e massima N. Possiamo fare alcune considerazioni sulla scelta del valore k, dimensione dell'array, più adeguato. Innanzitutto, sarà necessario che k sia maggiore di n: in caso contrario, non potremo immagazzinare l'interezza dei valori nemmeno nel caso migliore. Sarà poi conveniente scegliere un valore maggiore, tenendo comunque a mente che finchè k è minore di N avrò la sicurezza di avere collisioni. Dobbiamo però aspettarci che con numeri k simili a N avremo altà probabilità di collisione, e quindi la formazione di agglomerati, con successive ricerche inefficienti. Conviene quindi scegliere k più piccoli possibile, ma che non sacrifichino troppo le prestazioni della ricerca.

Inserzione dopo cancellazioni

C'è un'ottimizzazione possibile nel caso si vogliano effettuare cancellazioni: invece di assegnare al valore cancellato la chiave standard -1, possiamo assegnarvi una chiave diversa, sia -2. Nel caso vada a ricercare lo stesso elemento, una volta trovata la chiave -1 potrò direttamente arrendermi alla mia ricerca, in quanto sarò sicuro che l'elemento non sarà contenuto nella tabella. Possiamo quindi distinguere fra chiave:

- -1: elemento vuoto;
- -2: elemento disponibile.

E' esattamente per questo motivo che avevamo inserito la clausola:

```
if(A[x] == -1)
```

nella nostra funzione d'inserzione.

Scansioni
Esistono scansioni più efficienti della semplice scansione lineare. Vediamo una lista (non comprensiva):

- **Scansione lineare**: $h(x) + j \ \% \ k$;
- **Scansione quadratica**: $h(x) + j^2 \ \% \ k$, più efficiente (riduce gli agglomerati), ma rende nessario controllare di visitare tutte le posizioni dell'array, in modo da non fallire anche in caso di array non pieno.

Tempo medio di ricerca per l'indirizzamento aperto
Il tempo medio di ricerca (ovvero il numero di confronti) dipende da:

- **Fattore di carico**: il rapporto $\alpha = \frac{n}{k}$, compreso fra 0 e 1, che rappresenta il numero medio di elementi per ogni posizione;
- **Legge di scansione**: la scansione usata (meglio quadratica o ancora più sofisticata);
- **Uniformità della funzione hash**: la sua attitudine a generare indici con uguale probabilità.

Problemi con l'indirizzamento aperto
L'indirizzamento aperto non è sempre ottimale: molti inserimenti e cancellazioni degradano col tempo l'efficienza delle ricerche. Si rende necessaria una "rististemazione" periodica dell'array.

Hash con metodo di concatenazione
Vediamo un modo alternativo di implementare hash ovviando al problema delle collisioni. Possiamo associare ad ogni posizione dell'array una lista: eventuali collisioni risulteranno semplicemente in un inserzione in fondo alla lista della posizione trovata. Questo ci permette di scegliere un valore k minore di n (bound minimo di oggetti con cui avrò che fare), e evita del tutto gli agglomerati. Chiaramente, l'inefficienza è data dallo scorrimento lineare che dobbiamo fare quando andiamo a cercare valori che sono finiti in posizioni in fondo alla lista.

37 Programmazione dinamica

Finora abbiamo visto algoritmi divide et impera. Vediamo adesso un'altro tipo di approccio: la cosiddetta programmazione dinamica. La programmazione dinamica consiste nel risolvere i sottoproblemi dal basso e conservare

i risultati ottenuti per usarli successivamente (strategia bottom-up). Torna molto utile nel caso in cui non si sappia con esattezza in quali sottoproblemi dividere il nostro problema. La programmazione dinamica si può applicare quando il problema presenta:

- Sottostruttura ottima, ovvero una soluzione ottima del problema contiene una soluzione ottima dei sottoproblemi;
- Sottoproblemi comuni, ovvero gli stessi sottoproblemi ricorrono più volte nella soluzione secondo l'approccio ricorsivo.

Più lunga sottosequenze comune

Un'esempio di problema risolvibile da algoritmi di questo tipo è la ricerca della più lunga sottosequenza comune (PLSC). Poniamo di avere 2 stringhe: ci chiediamo di trovare il più grande sottoinsieme di caratteri (non necessariamente contigui, ma strettamente successivi) comune ad entrambe le stringhe. Siano due stringhe:

$$\alpha = \alpha_1, ..., \alpha_i, ..., \alpha_m, \quad \beta = \beta_1, ..., \beta_j, ..., \beta_n$$

E $L(i, j)$ la funzione che associa alle sottostringhe:

$$\alpha = \alpha_1, ..., \alpha_i \quad \beta = \beta_1, ..., \beta_j$$

La loro lunghezza di sottosequenza di lunghezza maggiore. Potremo allora stabilire la seguente relazione di ricorrenza:

- $L(0, 0) = L(i, 0) = L(0, j) = 0$
- $L(i, j) = L(i - 1, j - 1) + 1$ se $\alpha_i = \beta_j$
- $L(i, j) = \max(L(i, j - 1), L(i - 1, j))$ se $\alpha_i \neq \beta_j$

In pseudocodice, possiamo convertire la relazione di ricorrenza in un'algoritmo ricorsivo:

```
int length(char* a, char* b, int i, int j) {
  if(i == 0 || j == 0) return 0;
  if(a[i] == b[j])
    return length(a, b, i - 1, j - 1) + 1;
  else
    return max(length(a, b, i, j - 1),
               length(a, b, i - 1, j));
}
```

E trovare la complessità dalla (altra) relazione di ricorrenza:

$$T(k) = b + 2T(k-1)$$

che corrisponde a $O(2^n)$: l'algoritmo è particolarmente inefficiente. Vediamo se è possibile usare la programmazione dinamica. Abbiamo verificato l'esistenza di sottoproblemi comuni attraverso l'implementazione di un'algoritmo ricorsivo, vediamo allora se la sottostruttura del problema è ottima. Riprendiamo le stringhe:

$$\alpha = \alpha_1, ..., \alpha_i, ..., \alpha_m, \quad \beta = \beta_1, ..., \beta_j, ..., \beta_n$$

Intuitivamente, un certo sottoinsieme χ di α e β conterrà in sé una sottosequenza ottimale (di χ) che sarà a sua volta sottoinsieme della sottosequenza ottimale di α e β. Un'approccio in programmazione dinamica potrà allora essere implementato, e sarà quello di usare una matrice $(m+1) \times (n+1)$ (ci servono una riga e una colonna in più per le sottostringhe date da sottoinsiemi vuoti), dove precalcoleremo tutte le possibili combinazioni di i e j date dalla formula di ricorrenza. La matrice si evolverà quindi in complessità dal punto in alto a sinistra a quello in basso a destra, che rappresenterà il nostro risultato finale.

```
const int m = 7; const int n = 6;
int L [m+1][n + 1];

int quickLength(char* a, char* b) {
  for(int j = 0; j <= n; j++) {
    L[0][j] = 0; //azzero la prima riga
  }
  for(int i = 0; i <= m; i++) {
    L[i][0] = 0; //azzero la prima colonna
    for(j = 1; j <= n; j++) {
      if(a[i] != b[j])
        L[i][j] = max(L[i][j - 1], L[i - 1][j]);
      else
        L[i][j] = L[i - 1][j - 1] + 1;
    }
  }
  return L[m][n];
}
```

Dalla stessa rapprentazione di prima possiamo poi ottenere anche un esempio della sottosequenza maggiore, partendo dal risultato in fondo a destra e risalendo verso l'alto a sinistra fino a tornare all'origine (quindi trovando man mano gli elementi che hanno formato la sottosequenza maggiore, certo in ordine inverso):

```
void print(int** L, char* a, char* b, int i = m, int j = n) {
  if((i == 0) || (j == 0)) return;
  if(a[i] = b[j]) {
    print(a, b, i - 1, j - 1);
    cout << a[i];
  }
  else if (L[i][j] == L[i - 1][j])
    print(a, b, i - 1, j);
  else print(a, b, i, j - 1);
}
```

Con complessità peggiore O(n + m), nel caso in cui un "salto" sia necessario ad ogni iterazione. Notiamo che in ogni caso la complessità è lineare in quanto in ogni caso la ricorsione è unica per chiamata di funzione (solo un'alternativa viene scelta) e svolta su dimensione d'istanza diminuita di uno su almeno un'asse.

38 Algoritmi greedy

Un'algoritmo greedy (o algoritmo avido, goloso, curioso, ecc...) è un algoritmo che non fa sempre la scelta ottima (ottima globalmente), ma utilizza una certa euristica per fare la scelta migliore (ottima localmente). Ovviamente si richiede di adottare una buona euristica, ovvero una che fa corrispondere il più possibile la scelta ottima localmente a quella ottima globalmente. Sono algoritmi di tipo top-down.

Codici di compressione

Vediamo un esempio: abbiamo che un'alfabeto è un'insieme di caratteri, codificati in codice binario attraverso una certa stringa binaria (che può essere quella ASCII come quella Unicode, ecc...). La codifica di un testo significherà quindi la conversione di ogni carattere del testo nella sua corrispondente codifica in codice binario. La decodifica sarà allora la riconversione delle stringhe in codice binario in caratteri dell'alfabeto di partenza, che possano essere comprensibili ad un utente umano. Ci poniamo il problema di comprimere informazioni di questo tipo per ridurre lo spazio che occupano in memoria: questo può essere particolarmente utile nel caso tali informazioni debbano essere trasferite su qualche canale remoto, su rete, ecc... Questo può essere fatto, ad esempio, utilizzando codici a lunghezza variabile anziché fissa. Potremmo infatti decidere di assegnare codici di lunghezza maggiore a caratteri più rari, e di lunghezza minore a caratteri più frequenti: i caratteri più frequenti (e quindi la maggioranza del testo) richiederanno quindi un minore spazio di archiviazione. A questo punto è utile rappresentare i codici

a lunghezza variabile attraverso un'albero binario. Sia 0 codifica di un passo a sinistra, 1 codifica di un passo a destra, la stringa in binario

rappresenterà la sequenza di caratteri: Possiamo dire che l'albero ha tante foglie quanti sono i caratteri dell'alfabeto che consideriamo. La decodifica di una stringa significherà trovare un certo cammino lungo quest'albero. La codifica sarà ottimale quando l'albero sarà completamente binario.

Codici di Huffman

Questo algoritmo è alla base dei codici di Huffman, il cui obiettivo è quello di costruire una codifica ottimale (che minimizza la lunghezza delle codifiche dei testi). L'algoritmo presentato è un'algoritmo greedy e bottom-up rispetto all'albero (cioè che parte dai caratteri (foglie) per arrivare all'albero). L'algoritmo di Huffman gestisce una foresta di alberi: all'inizio del ciclo abbiamo n alberi di un solo nodo con le frequenze dei caratteri. Ad ogni passo, da lì in poi, vengono fusi i due alberi con radice minore introducendo una nuova radice avente come etichetta la somma delle due radici. Vediamo un'implementazione: la struttura dati migliore per poter fare operazioni del genere è un min-heap, ovvero un heap che mantiene il valore minimo alla radice. Il ciclo descritto avrà su questa struttura complessità di $O(\log n)$ su n iterazioni, ovvero $O(n \log n)$. L'approccio greedy funziona perchè ottimo locale e globale coincidono: se sistemiamo prima i nodi con frequnza più bassa, questi apparterranno a livelli più alti dell'albero.

39 Dizionari

Un dizionario è una struttura che si discosta dal concetto (adottato finora) di indirizzamento diretto. Conviene usare dizionari quando la dimensione del dominio degli input è di molto maggiore alla dimensione di archiviazinoe che abbiamo a disposizione. Il dizionario può essere implementato attraverso il meccanismo appena visto dell'hashing, e quindi della scrittura in memoria di coppie chiave-valore: questa struttura dati prende nello specifico il nome di *hash table* (tabella hash).

Hash table semplice

Sviluppiamo una tabella hash che gestisce interi maggiori di 0, usa chiavi coincidenti coi valori, e la funzione modulo come funzione hash. Per convenzione, prendiamo 0 come valore vuoto. Una possibile classe che implementa la tabella è:

```
1 class HashTable {
```

```
  int* _table;
  int _size;

public:
  HashTable(int size);
  bool insert(int key);
  void print();
  int hash(int key);
};
```

dove ci siamo dotati di un'array (in memoria dinamica) di interi, un intero rappresentante la dimensione dell'array, e le funzioni membro HashTable(), costruttore, insert(), che inserisce coppie chiave-valore (qua coincidenti) nella hash table, print() che stampa i contenuti della tabella, e hash, che è la funzione hashing stessa.

- **Costruttore**
  Il costruttore avrà la forma:

```
HashTable::HashTable(int size) {
  _table = new int[size];
  _size = size;
  memset(_table, 0, suze * sizeof(int));
}
```

  Notiamo l'uso della funzione di libreria memset(address, value, size), che serve ad inizializzare più velocemente la nostra array. La memset prende come argomenti un indirizzo, un valore, e una dimensione che determina il blocco di memoria contigua a partire dall'indirizzo da riempire col valore fornito (qui 0, che segnala l'elemento vuoto).

- **Hash**
  La funzione di hashing, come già detto, avrà la forma:

```
  int hash(int key) {
    return key % _size;
  }

```

- **Insert**
  La funzione dovrà, in ordine:
  - Trovare l'indice possibile tramite la funzione hash;
  - Controllare se la posizione è già occupata, e adoperare in tal caso misure preventive;
  - In caso contrario, inserire l'elemente assegnando la chiave all'indirizzo trovato.

Una possibile implementazione di questo algoritmo sarà:

```
bool HashTable::insert(int key) {
  int index = hash(key);
  if(_table[index] != 0)
    return false;
  _table[index] = key;
  return true;
}
```

Si palesa il problema di gestire eventuali collisioni. L'approccio appena utilizzato (quello di evitare completamente l'inserimento nel caso non sia possibile) non è infatti ottimale. Un'approccio possibile è quello di trasformare ogni elemento del vettore in una lista, dove gli oggetti in collisione verrano memorizzati successivamente uno dopo l'altro. Questo significherà trasformare il nostro vettore di interi in un vettore di puntatori ad intero, che farà da radice per una lista (volendo anche bidirezionale, agevola le eliminazione) di interi. Chiamiamo questa lista *lista di trabocco*. Mettiamo il tutto in uno struct:

```
struct Elem {
  int key;
  Elem* next;
  Elem* prev;
  Elem() : next(NULL), prev(NULL) {}
};
```

e a questo punto ridefiniamo la classe HashTable:

```
class HashTable {
  Elem** _table;
  int _size;

public:
  HashTable(int size);
  bool insert(int key);
  void print();
  void printOccupancy();
  int hash(int key);
  bool find(int key);
};
```

dove abbiamo aggiunto le funzioni printOccupancy(), che restituisce l'occupazione di ogni chiave del vettore; e la find, che va a cercare una chiave (eseguendo lo scorrimento linare della lista corrispondente alla sua chiave). Questo ci permette di riscrivere la insert come:

```
bool HashTable::insert(int key) {
  int index = hash(key);
```

```
  Elem* n = new Elem();
  n->key = key;
  n->next = _table[index];
  if(n->next != NULL)
    n->next->prev = n;
  _table[index] = n;
  return true;
}
```

e di scrivere la find come:

```
bool HashTable::find(int key) {
  int index = hash(key);
  Elem* pun = _table[index];
  while(pun != nullptr) {
    if(pun->key == key)
      return true;
    pun = pun->next;
  }
  return false;
}
```

Possiamo adesso fare una considerazione: potrebbe essere utile usare, invece di una lista, una struttura dati più avanzata che permetta di velocizzare gli scorrimenti dei singoli elementi del vettore, oppure semplicemente di mantenere l'ordinamento della lista in fase di inserimento. Notiamo però che, come sempre, un operazione di questo tipo richiede un tradeoff: possiamo velocizzare le ricerche, ma a costo di rallentare gli inserimenti.

Hashing di stringhe

Vediamo come realizzare questo tipo di struttura dati sulle stringhe. Per la funzione di hashing, potremmo pensare di usare la prima lettera:

```
int hash(string key) {
  return key[0] % _size;
}
```

Notiamo però che quest'approccio è assolutamente terribile per l'occupazione del vettore: molto probabilmente andremo a riempire esageratamente solo alcuni indici (quante frasi iniziano con la lettera x?). Un'approccio migliore potrebbe essere quello di sommare i valori di ogni carattere:

```
int hash(string key) {
  int index = 0;
  for(int i = 0; i < key.length(); i++) {
    index += key[i];
  }
```

```
6   return index % _size;
7 }
```

Che ci assicurebbe di avere una distribuzione più uniforme di chiavi.

Considerazioni sulla funzione di hash

Dalla funzione di hash si possono richiedere diverse caratteristiche per diverse applicazioni. In generale, possiamo parlare di:

- **Uniformità**, ovvero la copertura (o non) che ha la funzione di hashing per tutti i valori del dominio (nell'esempio precedente abbiamo visto come non è conveniente concentrare più chiavi sugli stessi indici).
- **Non reversibiilità**, ovvero la difficoltà di ricondurre un indirizzo alla sua chiave. Questo è particolarmente importante nell'ambito della sicurezza informatica.

In generale, si può anche dire che la maggior parte delle funzioni di hash lavorano sulla rappresentazione binaria delle chiavi.

La versione STL

La STL ci fornisce un'implementazione della funzione di hash: la cosiddetta stl::map:

```
1 stl::map <key_T, obj_T > table;
```

crea una tabella di oggetti di tipo obj_T con chiavi di tipo key_T. Potremo a questo punto fare le operazioni:

```
1 //inserimento
2 table['uno'] = "Valore uno";
3 //ricerca
4 table.find('uno');
```

40 Grafi

Grafi orientati

Un grafo orientato $g(N, A)$ è un'insieme di N nodi e di $A \subseteq N \times N$ archi. Un'arco è una coppia ordinata di nodi. Se $(p, q) \subseteq A$, diciamo che p è predecessore di q, e che q è successore di p. Definiamo poi $n = |N|$ numero dei nodi e $m = |A|$ numero degli archi. Un grafo orientato con n nodi ha al massimo n^2 archi. Un **cammino** su un grafo orientato è una sequenza di nodi $(n_1, n_2..., n_k), k \geq 1$ tale che esiste un arco da n_i a n_{i+1} per ogni $1 \leq i \leq k$. La lunghezza del cammino è data dal numero degli archi. Un ciclo è un cammino che comincia e finisce sullo stesso nodo. Un grafo dove non esistono cammini ciclici si chiama aciclico.

Rappresentazioni dei grafi

Possiamo rappresentare i grafi in più modi:

- **Liste di adiacenza**
  Una lista di adiacenza è un modo di rappresentare i grafi attraverso una lista di nodi accessibili da ogni nodo, cioè:

```
struct Node {
  int nodeNumber;
  Node* next;
};
Node *graph[N]; //radice del grafo
```

  Si definisce un'array con un dimensione uguale al numero N di nodi. Ogni elemento dell'array è a questo punto uno struct che rappresenta un nodo e i suoi successori.

- **Matrici di adiacenza**
  Un'altro tipo di rappresentazione è attraverso le matrici di adiacenza. Si realizza una matrice quadrata $n \times n$ che rappresenta il prodotto cartesiano dei nodi. L'elemento della matrice i, j sarà a questo punto 1 se esiste un arco dal nodo i al nodo j e 0 altrimenti: questo tipo di struttura dovrebbe essere già stata studiata ad algebra lineare. Si ricorda che la potenza n-esima di tale matrice rappresenta per ogni coppia di nodi il numero dei cammini di lunghezza n fra di essi.

Vediamo come gestire le informazioni immagazzinate dai grafi, attraverso le **etichette**:

- **Liste di adiacenza etichettate**
  Assegnamo un'etichetta sia ai nodi che agli archi:

```
struct Node {
  int nodeNumber;
  ArcType arcLabel;
  Node* next
};
Node* graph;
NodeType nodeLabels[N];
```

  Potremo allora immagazzinare un valore di tipo ArcType per ogni arco, e un valore di tipo NodeType per ogni nodo. Il NodeType di un nodo si potrà trovare per corrispondenza diretta con gli indici del vettore di liste, mentre l'ArcType sarà associato ad ogni elemento della lista di adiacenze del nodo.

- **Matrici di adiacenza etichettata**
  Per le matrici di adiacenza etichettate, possiamo usare un'approccio identico per i nodi, e indicare l'ArcType direttamente sulla matrice.

Visità in profondità

Vediamo come effettuare una visita su una struttura dati grafo: la visita in profondità. Nella visita in profondità gli archi vengono esplorati a partire dall'ultimo nodo esaminato che abbia ancora degli archi non esplorati uscenti da esso. Questo comportamento viene implementato scorrendo tutti i nodi, marchiando man di mano quelli che sono già stati visitati. I nodi appena visitati vengono "scorsi" fino in fondo dove non si trovano nodi già marchiati, e solo dopo si passa al nodo successivo. Definiamo la classe grafo basata sulle liste di adiacenza:

```
class Graph {
  struct Node {
    int nodeNumber;
    Node* next;
  };
  Node* graph[N];
  NodeType nodeLabels [N];
  int mark[N];
};
```

Il vettore mark[] rappresenta il marchio dato ai nodi visitati. A questo punto la visita in profondità potrà essere implementata come:

```
void nodeVisit(int i) {
  mark[i] = 1;
  //esamina nodeLabels[i]
  Node* q; int j;
  for(q = graph[i]; g; g = g->next) {
    j = g->nodeNumber;
      if(!mark[j]) nodeVisit(j);
  }
}

public:
void depthVisit() {
  for(int i = 0; i < N; i++)
    mark[i] = 0;
  for(i = 0; i < N; i++) {
    if(!mark[i])
      nodeVisit[i];
  }

}
```

Grafi non orientati

Un grafo non orientato $g(N, A)$ è una struttura non dissimile dal grafo orientato, ma dove le coppie che rappresentano gli archi non sono ordinate. Se $(p, q) \subseteq A$, diciamo che p è adiacente a q e viceversa. Un grafo non orientato con n nodi ha al massimo $\frac{n(n-1)}{2}$ archi. Un **cammino** su un grafo non orientato è una sequenza di nodi $(n_1, n_2..., n_k), k \geq 1$ tale che esiste un arco fra n_i a n_{i+1} per ogni $1 \leq i \leq k$. La lunghezza del cammino è data dal numero degli archi. Un ciclo è un cammino che comincia e finisce sullo stesso nodo, e non ha ripetizioni, eccetto per l'ultimo nodo. Un grafo non orientato è connesso se esiste sempre un cammino fra due nodi qualsiasi del grafo.

Rappresentazione dei grafi non orientati

Un grafo non orientato può essere rappresentato attraverso le tecniche viste per i grafi orientiati, assumendo per ogni arco dal nodo a al nodo b, un'arco inverso dal nodo b al nodo a. Naturalmente ogni arco del grafo non orientato sarà rappresentato due volte (nel caso della matrice di adiacenza, essa sarà simmetrica).

Multi-grafi orientati e non

Un multigrafo $m(N, A)$ è un'insieme di N nodi e di A insiemi di archi su qualsiasi nodo. In sostanza, un multi-grafo elimina la dipendenza fra il numero di nodi e il numero di archi. I multi-grafi possono essere orientati e non, a seconda se le coppie che formano gli archi sono ordinate o no.

Minimo albero di copertura

Sappiamo che un grafo non orientato è connesso se esiste un cammino fra due nodi qualsiasi del grafo. Una **componente connessa** è un sottografo connesso del grafo. La componente connessa *massimale* è una componente connessa che non è contenuta in nessun'altra componenete connessa, ovvero una componente connessa che non è connessa a nessun'altro nodo tramite un'arco addizionale. A questo punto, un **albero di copertura** è un insieme di componenti connesse massimali acicliche, e il minimo albero di copertura è l'albero di copertura che ha somma dei pesi degli archi minima.

Algoritmo di Kruskal per il minimo albero di copertura

L'algoritmo di Kruskal trova il minimo albero di copertura di un grafo attraverso un'algoritmo greedy. Il processo dell'algoritmo si può ridurre a:

- Ordina gli archi del grafo in ordine crescente;
- Scorri l'elenco degli archi. Per ogni arco a:
  - Se a connette due componenti non connesse, scegli a e unifica le componenti.

Algoritmo di Dijkstra

L'algoritmo di Dijkstra serve a trovare i cammini minimi da un nodo di partenza a tutti gli altri nodi, su un grafo con archi pesati. Si tratta di un algoritmo greedy. Si adoperano due tabelle, che chiameremo *dist* (distanza) e *pred* (predecessore) con n elementi pari al numero dei nodi. Per ogni nodo a, dist(a) contiene in ogni momento la lunghezza di un cammino dal nodo iniziale ad a e pred(a) il predecessore di a in questo cammino. I nodi sono divisi in due insiemi: quelli già sistemati, ovvero per i quali i valori delle tabelle *dist* e *pred* sono già stati decisi definitivamente, e quelli da sistemare (che chiameremo insieme Q). Si noti che anche i nodi nell'insieme Q hanno valori in *dist* e *pred*, relativi al percorso migliore trovato fino a quel momento. Il processo può essere descritto come:

- Considera come "sistemato" il nodo con *dist* minore in Q e rimuovilo;
- Aggiorna *dist* e *pred* per ogni nodo immediatamente successore del nodo appena rimosso;
- Ripeti finche Q non è vuoto.

In pseudocodice molto approssimativo, possiamo dire:

```
1  Q = N;
2  per ogni nodo p diverso da p_0 {
3    dist(p) = infty, pred(p) = vuoto;
4  }
5  dist(p_0) = 0;
6  while(Q contiene piu di un nodo) {
7    estrai da Q il nodo p con minima dist(p);
8    per ogni nodo q successore di p {
9      lpq = lunghezza di arco (p, q);
10     if(dist(p) + lpq < dist(q)) {
11       dist(q) = dist(p) + lpq;
12       pred(q) = p;
13       reinserisci q in Q;
14     }
15   }
16 }
```

La struttura dati più adatta per Q è chiaramente un min-heap: vogliamo ad ogni iterazione restituire l'elemento minore. Valutiamo la complessità dell'algoritmo di Dijkstra implementato con un min-heap. Abbiamo che il ciclo while effettua n iterazioni, e che ogni iterazione costa:

$$C = C[L_7] + \frac{m}{n} C[L_{15}]$$

ovvero $O(\log n + \frac{m}{n} \log n)$. La complessità finale sarà quindi:

$$O(n(\log n + \frac{m}{n} \log n)) = O(n \log n + m \log n)$$

complessità lineare logaritmica (linearitmica).

Applicazioni di algoritmi sui grafi
Vediamo alcuni esempi di applicazione dei grafi e degli algoritmi su di essi definiti.

- **Graph coloring**
  Si ha un problema di etichettamento dei grafi: dato un numero K di colori da associare ad ogni nodo del grafo, si trovi una colorazione che assicura che nessun nodo abbia lo stesso colore di un suo adiacente. Un'esempio di problemi di questo tipo può essere semplicemente il gioco del Sudoku.

- **Algoritmo PageRank di Google**
  L'algoritmo PageRank di Google serve al motore di ricerca per trovare le pagine web di interesse per una interrogazione. Cerca di stimare il comportamento di un "*random surfer*" che naviga il web basandosi sulle connessioni (*link*) fra le pagine. Riesce a fare questo considerando la rete come un grafo (*webgraph*) in cui le pagine sono i nodi, e i link gli archi. Su ogni nodo (quindi per ogni pagina web) P viene calcolato un "rango", che ne rappresenta la rilevanza, $R(P)$. $R(P)$ dipende da quanti link arrivano a e provengono da P:

  $$R(P) = \sum_{Q \to P} \frac{R(Q)}{|Q|}$$

  dove $|Q|$ è il numero di link uscenti da Q, e $Q \to P$ i link da Q a P. Questo calcolo viene fatto ottenendo iterativamente il rango dei nodi utilizzando la matrice di adiacenza e partendo da un valore del rango uguale per tutti i nodi.

- **Graph database**
  I database basati sui grafi forniscono un'alternativa al modello relazionale, sopratutto nei casi in cui le tabelle siano troppe e troppo grandi. Secondo questo approccio, i vertici rappresentano tabelle e gli archi relazioni.