huffdiff

attenzione: image_watch vuole le immagini a colori con schema BGR non RGB

```
frame(r,c) = {r,g,b};
im_frame(r,c) = {b,g,r};
```

Note

- Per leggerlo con segno, lo leggo senza segno e poi faccio il casting in intero con segno

```
uint32_t u;
i = static_cast<int32_t>(u);
```

- for_each prende il primo elemento della matrice e chiama f (freq_counter) con l'operatore parentesi; il for each vuole f per copia, lo popola (alterandolo) e lo ritorna. Grazie ad un assegnamento vado quindi a salvare il progresso fatto
- Calcolo poi l'entropia delle differenze (che è naturalmente minore a quella della rana integrale)
- Abbiamo definito l'oggetto huffman a template così per poterlo usare per ogni genere di tipo.

```
huffman<int> h; // creo oggetto

h.create_table(f); // passo le frequenze, l'unica cosa di cui ho bisogno

h.generate_canonical_codes(); // genero i codici canonici
```

- scrittura

```
h.codes_table_.size() // numero di entries, attributo privato

for(auto& x: h.codes_table_){
 bw(scrivi simbolo, 9 bit);
 bw(scrivi len, 5 bit);
}
```

- search map

```
associa al simbolo il codice in modo direto
```

Come è fatto huffman?

Internamente usa dei nodi

```
struct node{
  T sym_;
  size_t prob_;
  node* l_;
  node* r_;
}
```

Costruttori

il primo prende la singola probabilità

```
node(T& sym_, size_t prob){

}
```

prende due nodi

```
node(node*a, node*b){

}
```

pnode

perchè usare pnode? Per far funzionare un oggetto come un puntatore, si deve definire l'operatore `*` (asterisco)

```
struct pnode{
        node p_;

        pnode(node*p) : p_(p){};
        operator node*(){ return p_; }
}
```

permette di poter ordinare i puntatori ai nodi

code

L'oggetto code contiene i codici binari di huffman

create_table

Prende in input una mappa

-> Processo di delete dei puntatori
serve un vettore di unique_ptr a nodi chiamato storage, questo è un contenitore di oggetti che contengono i puntatori, in modo che ogni volta che creo un nodo lo pongo anche nello storage, quando la funzione termina distruggo il vettore, ciò funziona perchè il distruttore di un vettore chiama il distruttore di tutti gli oggetti all'interno nonchè di tutti i nodi presenti. (no memory leak) In questo modo quando distruggo il vettore distruggo gli oggetti.

La mappa ha first (simbolo) e second (probabilità), per ogni elemento della mappa creo un nodo e questo nodo lo pongo prima nel vettore di storage (così che diventa lui il proprietario) e poi in `vec`.

Faccio la sort di `vec` ed ordinerà per probabilità (siccome pnode definisce l'operatore su probabilità)

algoritmo huffman

finchè nel vettore ho elementi
prendi l'ultimo e toglilo dal vettore
prendi il penultimo (ora ultimo)
crea un nuovo nodo
facoltativo: salviamolo anche in storage
riaggiungi il (`in` itera in tutto il vettore, ha complessità O(n), se il vettore è ordinato serve la **ricerca binaria**), questo perchè container come la lista non danno alcuna garanzia sull'ordinamento, esempio:
```
1 5 7 7 7 9 10
```
in questo caso conviene una ricerca binaria non lineare
oltre a ciò, poniamo che vogliamo cercare 7 ma quale dei tre dovremmo prendere? primo, medio o ultimo? Se queste sono le probabilità, dove vogliamo inserire la nuova coppia di nodi che ha prob = 7?
1 5 *7 7 7 $ 9 10*
dove sta il dollaro o l'asterisco? Essendo un albero di huffman sappiamo che la lunghezza media è la migliore possibile (nonchè minima)
Cambia però la varianza delle lunghezze della codifica, se scegliamo il primo posto possibile (sotto = max varianza, sopra = minimizziamo la varianza) quindi in questo esempio ha max varianza possibile e $ minima varianza possibile. Noi vogliamo la varianza minima possibile siccome i codici più lunghi li fondiamo con gli altri più tardi possibili e sono più uniformi.

```
        *Grana crasha*

        1 5 * 7 7 7 $ 9 10
        sulla ricerca lineare possiamo semplicemente porre il 7 prima o dopo in base al primo o secondo che troviamo,
con la ricerca binaria invece è estremamente puntuale ed error-prone questa cosa.
        Grazie a ciò abbiamo definite lower_bound ed upper_bound, la lower_bound che usiamo per huffman fa la ricerca
```

```
binaria su una sequenza ordinata(non ha senso con le liste siccome non abbiamo l'accesso casuale O(1) ma comunque deve tornare indietro, meglio la lineare a quel punto)
        ```
```

- Inseriamo nella posizione trovata nel vettore
- Poniamo come root il nodo infine

## Calcolo lunghezze

Per calcolare le lunghezze serve una ricerca ricorsiva

Nota: se left è nullptr allora anche a destra lo è, questo tipo di albero ha un nome che grana non ricorda (non è completo, non bilanciato boh)

Procedo ricorsivamente a sx e dx aumentando la lunghezza di 1, notiamo che non calcoliamo i codici siccome è canonico

Pongo tutto nella codes_table che è un `vector<code>` cioè un vettore di codici

## Come si calcolano i codici canonici

- Prendo il valore corrente
- Lo shifto a sx per quanti bit mi servono (se la lunghezza è 3 e la corrente è 1 devo shiftare di 2 posizioni)
- Ottengo così il nuovo codice

### Esempio popolazione dei codici canonici:

nota: noi durante il debug vediamo i valori in decimale, dobbiamo fare quindi riferimento alla colonna `Len` che è ciò che l'exp. watch ci mostra

nota: lo shift va fanno per differenza o iterativo, nel senso che va shiftato dalla mia lunghezza fino a quella a cui devo arrivare

| L | Codice | Val | Len | Commenti |
|---|---|---|---|---|
| 2 | 00 | 0 | 2 | |
| 3 | 010 | 2 | 3 | (non sarebbe lungo abbastanza, quindi sommo 1 e shifto) |
| 3 | 011 | 3 | 3 | |
| 4 | 1000 | 4 | 4 | |
| 5 | 10010 | 18 | 5 | |
| 5 | 10011 | 19 | 5 | |
| 5 | 10100 | 20 | 5 | |
| 5 | 10101 | 21 | 5 | |

## Decodifica

**Attenzione**: nella read non si può scrivere (vedi giù) siccome la stringa non è un vettore di carattere bensì è un vero e proprio oggetto; questo è un approccio utilizzabile in C, ma in C++ dipende dall'implementazione.

Con in primo modo andiamo a rasare totalmente tutto l'oggetto string, noi vogliamo invece iniziare a scrivere all'indirizzo del primo oggetto del puntatore al carattere

Nota: E' necessario chiamare il costruttore siccome mi va ad inizializzare la stringa, altrimenti avrei un problema di memoria

NO:

```
string MagicNumber(8, ' ');
raw_read(in, Magic Number, 8);
```

OK:

```
string MagicNumber(8, ' ');
raw_read(in, MagicNumber[0], 8);
```

NO:
in questo modo non stiamo dicendo QUANTO leggere
```

```
string MagicNumber;
in >> MagicNumber;
```

- Per ogni table_entries vado a leggere simbolo e lunghezza, nota che abbiamo due read diverse, una per gli unsigned ed una per gli int con segno
- La push_back va ad accodare il codice alla tabella dei codici (nelle tabelle non abbiamo anche i codici siccome nel canonico vanno comunque ricalcolati)

```
h.codes_table_.push_back();
```

- Parto dall'inizio della tabella (i codici più corti che ci sono), se la lunghezza è maggiore del n_bit letti finora leggo un bit e lo aggiungo al codice; sposto a sx e aggiungo il bit. E anche possibile fare una lettura di n_bit per differenza di lunghezza tra i 2 come abbiamo fatto nella codifica (così da rimuovere il while a riga 122).
- In teoria il do-while a riga 132 e l'if a 133 non dovrebbero mai accadere, corrispondono ad un while(1), probabilmente sono inutili ed impossibili per costruzione per l'albero di huffman

Ricostruzione immagine:

- Sommo il valore precedente dalla matrice delle differenze
- Quando sono a fine riga quello superiore sarà l'ultimo della riga superiore