22/23

Algorithm engineering

ANALYSIS WITH PSEUDOCODES AND THEOREMS WITH PROOFS

GIUSEPPE LOMBARDI

UNIVERSITA' DI PISA

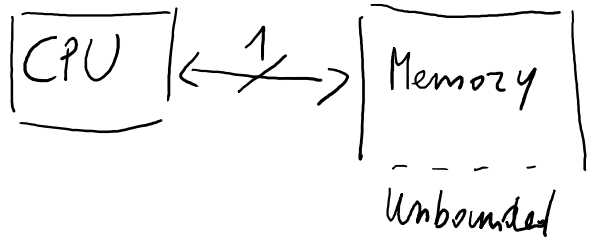
Table of contents

2-level memory model (disk model)

venerdì 27 gennaio 2023 15:32

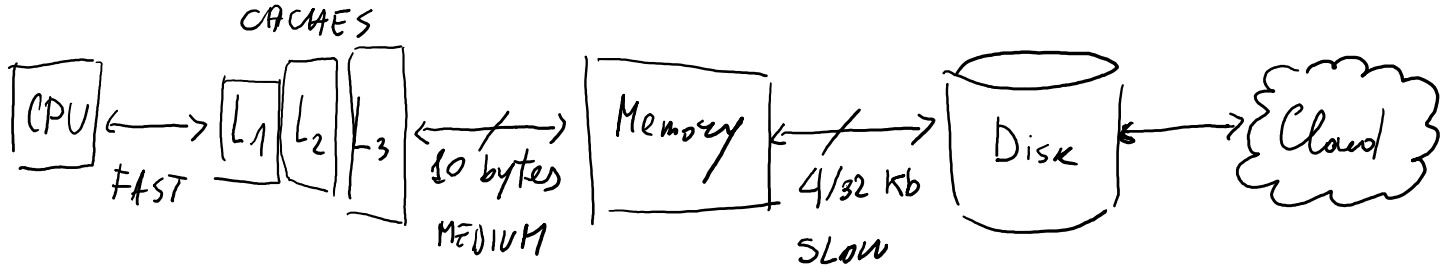
#I/Os good estimator for time complexity

Von Neumann Model (RAM model)

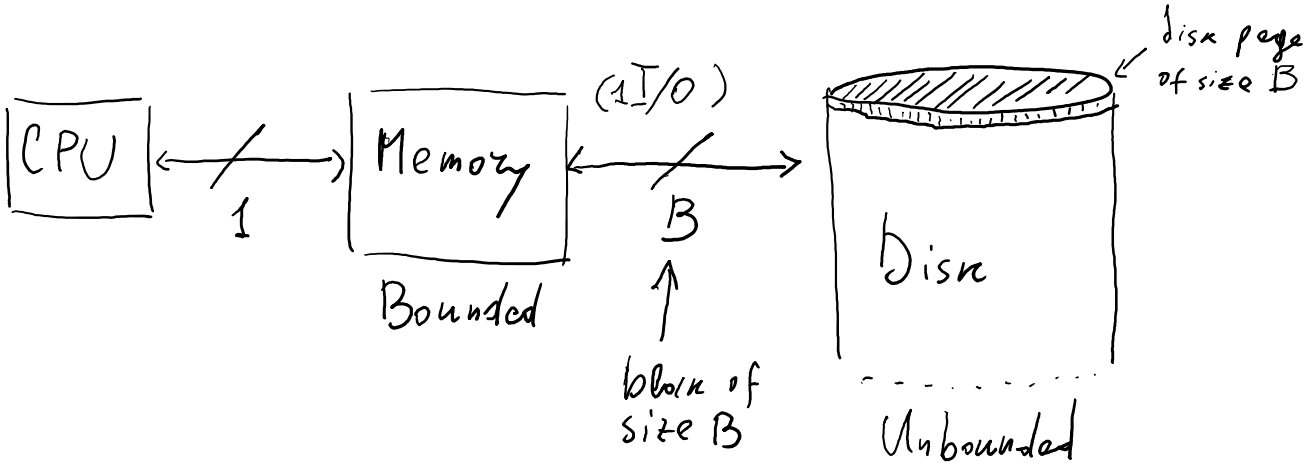


T(n) = # steps for input size n
(asymptotic and worst case)

Modern machine (hierarchy of memory levels)



2-level Memory model

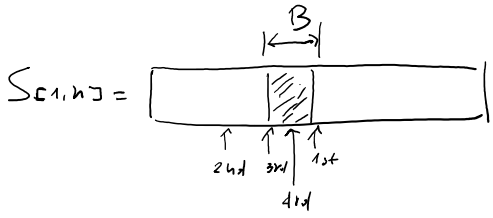


T(n) = #I/Os

B-tree

venerdì 3 marzo 2023 19:33

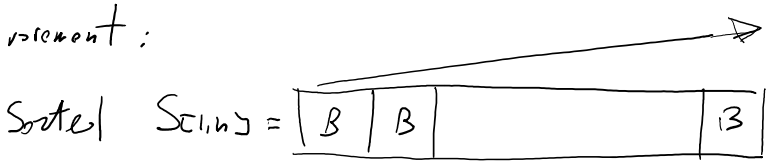
Binary Search



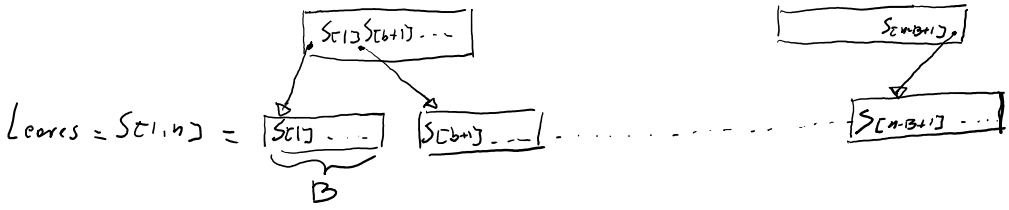
$T(n) = O(\lg_2 n)$

We expect $O(\lg_2 n) - \underbrace{O(\lg_2 B)}_{\text{binary search in subseq. of size} \le B} I/Os = O(\lg_2 \frac{n}{B}) I/Os$

An improvement:



B^+ tree



$$\text{FAN-OUT} = \frac{B}{\text{size(key)} + \text{size(pointer)}} = \Theta(B)$$

$$\#\text{LEAVES} = n/B$$

$$\Big\} \Rightarrow \text{high} = \lg_B \frac{n}{B}$$

```
SEARCH(B+-tree):
    node ← ROOT(B+-tree);
    while (node ≠ φ) {
        k ← BINARYSEARCH(node); // 1 I/O (same page)
        node ← CHILDREN(node, k); // go down to k-th children
    }
```

- Extra space:
- Time: $(\lg_B \frac{n}{B}) \cdot \lg_2 B = \frac{\lg_2 \frac{n}{B}}{\lg_2 B} \cdot \lg_2 B = \lg_2 \frac{n}{B}$ (BS: $\lg_2$)
- I/Os: $\lg_B \frac{n}{B}$ (= high of tree)

Usually: $B = 32kb = 2^5 \cdot 2^{10}$ byte $= 2^{15}$ byte

key/pointer $= 2^3$ byte

$\lg_B \frac{n}{B}$ I/Os $= \frac{\lg_2 \frac{n}{B}}{\lg_2 B} = \frac{\lg_2 \frac{n}{B}}{15}$. Assume n = 1 terabyte $= 2^{40}$ byte

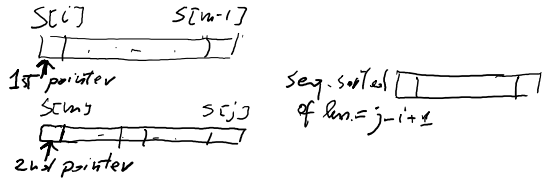
ca $2^{40}/2^{15}$

$$\frac{\lg_2 B}{15}$$

$$\Rightarrow \frac{\lg_2 2^{40/2^{15}}}{15} = \frac{25}{15} < 2 \quad \text{ok !!}$$

Merge-based sorting paradigm

lunedì 13 febbraio 2023 20:04



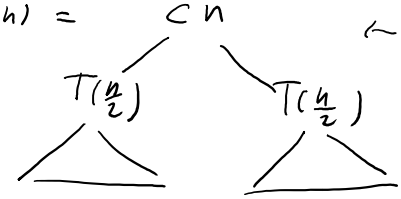
$S[1,n] = s_1, \dots, s_n$

```
MERGESORT(S, i, j)
if i<j {
    m = (i+j)/2;
    MERGESORT(S, i, m-1);
    MERGESORT(S, m, j);
    MERGE(S, i, m, j);
}
```

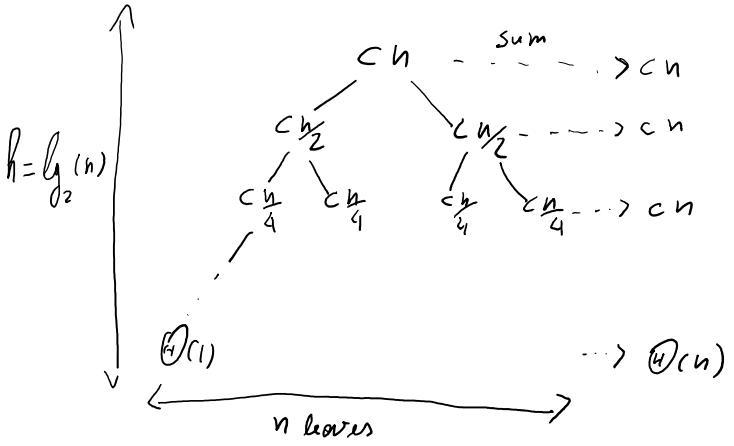
- Extra space: $O(n)$ (aux. array for merging)
- Time: Merge sort $(1, n)$

$$T(n) = 2T(n/2) + \underbrace{O(n)}_{\text{Merging}} \overset{(*)}{\Rightarrow} O(n \lg n)$$

(*) $T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{if } n>1 \end{cases}$

Recursion tree: $T(n) =$ 

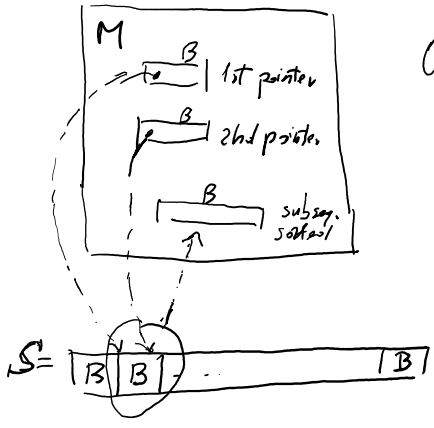
← the cost of this algorithm is the sum of the cost of all the node in the recursive tree



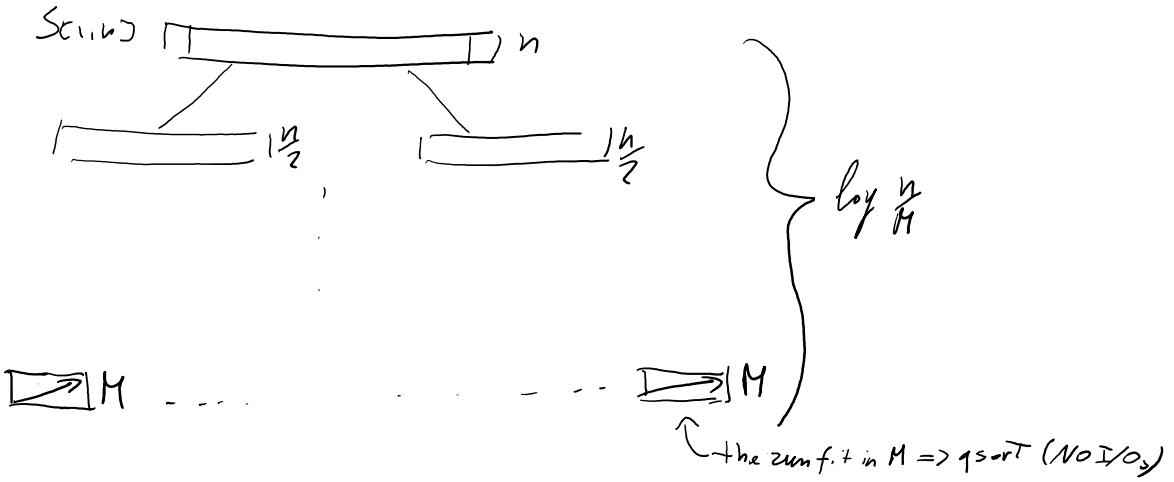
Total cost $= \underbrace{\lg_2 n}_{h} \cdot cn + \Theta(n) = cn \lg_2 n + \Theta(n) = \Theta(n \lg_2 n)$

Disk Model ($n > M$)

Suppose $M \geq 3B$



$O(\frac{n}{B})$ I/Os to merge 2 subseq. of total size $= n$



#seq. merged at a time

#runs (size of runs = M)

• I/Os: $O\left(\frac{n}{B}\log_{(2)}\frac{n}{(M)}\right)$ $\left(= O\left(\frac{n}{B}\log_2 n\right) - O\left(\frac{n}{B}\log_2 M\right)\right)$

I/Os for merging — #levels — I/Os saved

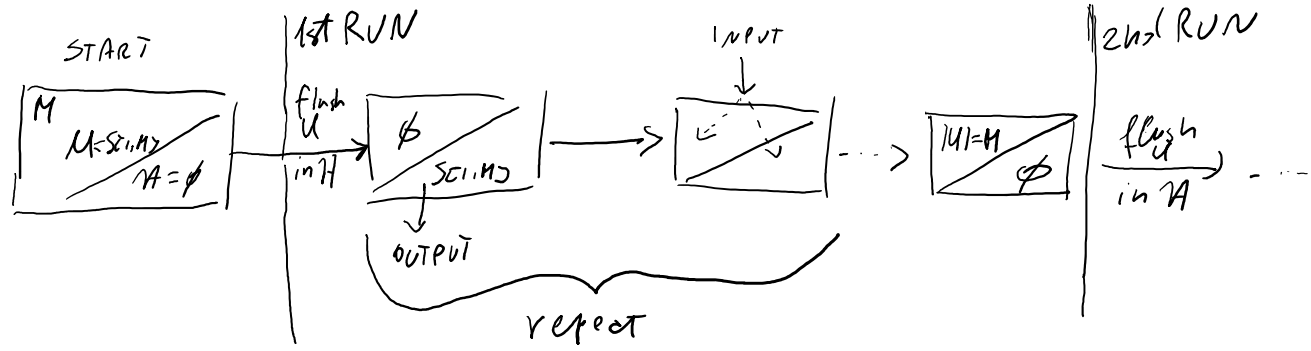
(1st RUN)

```
SNOWPLOW(S, M):
    M ← S[1,M];
    H ← MIN-HEAP(M);
    M ← ∅;
    while (H ≠ ∅) {
        min ← POP(H);
        OUTPUT(min);
        next ← NEXT(S);
        if (next ≥ min)
            U ← ADD(U, next);
        else
            H ← INSERT(H, next);
    }
```

S[1,n], U = set, H = min-heap

- Extra space: NO (array-based impl. for Heap)
- I/Os: $O\left(\frac{n}{B}\log_2 \frac{n}{(2M)}\right)$ (see Prop)

Avg. size of runs.



Prop Average length of runs is $\gamma = 2M$, supposing random distribution

Proof γ items of a run go in U or H. But in U they will not removed in a run.

$\tau = M + (\tau - M)$, Random distr. $\Rightarrow p(\text{next} < \min) = \frac{1}{2}$

(↑ inserted in $\mathcal{U}$, ↑ inserted in $\mathcal{A}$)

$\Rightarrow E[\text{items put in } \mathcal{U}] = \frac{1}{2} \cdot \tau = M$

$\Rightarrow E[\text{size run}] = \tau = 2M$ ▨

Cor $p(\text{next} < \min) = \alpha \ (0 < \alpha < 1) \Rightarrow \tau = \frac{M}{\alpha}$

Suppose $M \gg 3B$, and $K = \left(\frac{M}{B} - 1\right) \gg 2$ block available (↑ aux. space)

START: K ordered subseq. $R_1, \dots, R_K$ (K distinct runs)

$\mathcal{A}$ min-Heap containing $\langle R_1[1], 1 \rangle, \dots, \langle R_K[1], K \rangle$

(↑ smallest item, ↑ run)

```
MULTI-WAY MERGESORT(R_1, ..., R_K):
    x_1 ← 1; ...; x_K ← 1;
    A ← MIN-HEAP((R_1[x_1], 1), ..., (R_K[x_K], K));
    while (A ≠ φ) {
        (min, run) ← POP(A);
        OUTPUT(min);
        x_run++;
        if x_run ≤ LEN(R_run)
            A ← INSERT(R_run[x_run], run);
    }
```

- Extra space: NO
- Time/#Comparisons: $O(n \log n)$
- I/Os: $O\left(\frac{n}{B} \log_{\left(\frac{M}{B}\right)} \frac{n}{M}\right) = O\left(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B}\right)$

($\left(\frac{M}{B}\right)$ = # seq (runs) merged at time)

$\log_{\frac{M}{B}}(M) = \log_{\frac{M}{B}}\left(\frac{M}{B} \times B\right)$

$= 1 + \log_{\frac{M}{B}}(B)$

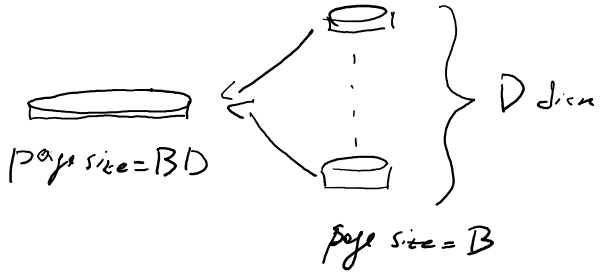
$= \Theta\left(\log_{\frac{M}{B}}(B)\right)$

Disk striping

mercoledì 1 marzo 2023 15:46

Using $D>1$ disks to increase the bandwith of the I/O. Simplest approach is to consider one big disk of page size $B' = DB$

(DB must be $< M$)



Using the multi-way mergesort the I/O complexity is

$$O\left(\frac{n}{DB}\log_{\frac{M}{DB}}\frac{n}{M}\right) > \Omega\left(\frac{n}{BD}\log_{\frac{M}{B}}\frac{n}{M}\right)$$

↑ MULTI-WAY MERGESORT ↑ LOWER BOUND

Analysing the ratio r:

$$r = \frac{\frac{n}{BD}\log_{\frac{M}{BD}}\left(\frac{n}{M}\right)}{\frac{n}{BD}\log_{\frac{M}{B}}\left(\frac{n}{M}\right)} = \frac{\frac{\log_2\left(\frac{n}{M}\right)}{\log_2\left(\frac{M}{BD}\right)}}{\frac{\log_2\left(\frac{n}{M}\right)}{\log_2\left(\frac{M}{B}\right)}} = \frac{\log_2\left(\frac{M}{B}\right)}{\log_2\left(\frac{M}{BD}\right)} = \frac{\log_2\left(\frac{M}{B}\right)}{\log_2\left(\frac{M}{B}\right) - \log_2(D)}$$

$$= \frac{1}{1 - \frac{\log_2(D)}{\log_2\left(\frac{M}{B}\right)}} = \frac{1}{1 - \log_{\frac{M}{B}}(D)}$$

Conclusion: Multi-way Mergesort is not optimal with $D>1$ and the ratio r is $r>1$, r is high if $D \uparrow \frac{M}{B}$ $\left(1 < D \leq \frac{M}{B}\right)$

Permuting

mercoledì 1 marzo 2023 11:28

$\pi[1,n] = \pi(1), \dots, \pi(n)$ (i.e. $\pi[1]$, $\pi[2]$, ..., $\pi[n]$)

$S = S[1,n]$ $\quad \langle S, \pi \rangle \longrightarrow S[\pi(1)], \dots, S[\pi(n)]$

- RAM MODEL:

```
PERMUTING_RAM(S,π):
    S' = S;
    for i=1,...,n
        S'[i] = S[π[i]];
    return S'
```

time: $O(n)$

- DISK MODEL:

PERMUTING1_DISK(S,π) mimicks PERMUTING_RAM(S,π)

- I/Os: $O(n)$

```
PERMUTING2_DISK(S,π):
    P ← <1,π(1)>,...,<n,π(n)>;
    P ← SORT(P,2); // sort on 2nd component
    for i=1,...,n
        P[i,2] ← S[i]; // P = <i1,S[1]>,...,<in,S[n]>
    P ← SORT(P,1); // sort on 1st component
    return P[:,2];
```

- I/Os: $\underbrace{2\frac{n}{B}}_{\text{2 scans}} + \underbrace{2\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{M}}_{\text{2 sorts}}$

$= O\left(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{M}\right)$

Thm Permuting n items takes $O\left(\min\left\{n, \frac{n}{B}\log_{\frac{M}{B}}\frac{n}{M}\right\}\right)$ I/Os on disk model

Coz On Disk Model I/O complexity of sorting and of permuting are equal iff $\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{M} < n \iff B > \log_{\frac{M}{B}}\frac{n}{M}$, so always true in pratics

Upper bounds

mercoledì 1 marzo 2023 12:44

	RAM model	Disk Model	
Permuting	$O(n)$ SCANNING	$O\left(\min\left\{n, \frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B}\right\}\right)$	SEE THM of permuting
Sorting	$O(n \log n)$ MERGE SORT	$O\left(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{M}\right)$	MULTI-WAY MERGESORT

In case $D > 1$

	Disk model	
Sorting	$O\left(\frac{n}{BD}\log_{\frac{M}{BD}}\frac{n}{M}\right)$	MULTI-WAY MERGESORT with DISK STRIPING TECNIQUE

Lower bound on sorting

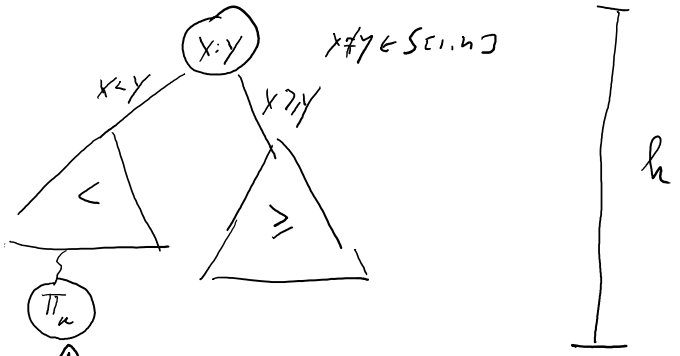
mercoledì 1 marzo 2023 15:00

- RAM model (Comparison-based sorting)

A binary tree for each input size: each node represents a comparison between two items

$S[1..n]$

FAN-OUT = 2



leaf is a permutation (a particular solution for a specific input of size n)

We must have $2^h \geq n!$ $\Leftrightarrow$ $h \geq \lg_2(n!) \approx n \lg_2(n)$

$n!$: all possible permutations (leaves)

Stirling's approximation

Prop time complexity lower bound (RAM model) for comparison based sorting is $\Omega(n \lg_2 n)$

- Disk Model

A node in the comparison tree is an I/O (so reads B items from disk)

FAN-OUT = # permutations generated by the I/O

$$\text{FAN-OUT} = \begin{cases} \binom{M}{B} B! & \text{for new items} \\ \binom{M}{B} & \text{for old item (just fetched)} \end{cases}$$

leaves = $n!$ (all possible permutations)

The #I/Os accessing to new items must be $\frac{n}{B}$. So assuming

$$t = \frac{n}{B} + \left(t - \frac{n}{B}\right)$$

all I/Os — "new" I/Os — "old" I/Os

We must have:

$$\left[\binom{M}{B} B!\right]^{n/B} \cdot \binom{M}{B}^{t-n/B} \geq n!$$

$$\lg\left(\left[\binom{M}{B} B!\right]^{n/B} \cdot \binom{M}{B}^{t-n/B}\right) \geq \lg(n!)$$

$$\frac{n}{B}\lg\left(\binom{M}{B} B!\right) + \left(t - \frac{n}{B}\right)\lg\binom{M}{B} \geq n \lg n$$

$$\frac{n}{B}\left(B\lg\left(\frac{M}{B}\right) + B\lg B\right) + \left(t - \frac{n}{B}\right) B \lg\left(\frac{M}{B}\right) \geq n \lg n$$

$$n \lg B + tB \lg\left(\frac{M}{B}\right) \geq n \lg n$$

$$t \geq \frac{n \lg n - n \lg B}{B \lg\left(\frac{M}{B}\right)} = \frac{n \lg\left(\frac{n}{B}\right)}{B \lg\left(\frac{M}{B}\right)} = \frac{n}{B} \lg_{\frac{M}{B}}\left(\frac{n}{B}\right)$$

Col 1 Lower bound for comparison-based sorting in disc-model is $\Omega\left(\frac{n}{B} \lg_{\frac{M}{B}}\left(\frac{n}{B}\right)\right)$ I/Os

If $D > 1$ disks $\Omega\left(\frac{n}{BD} \lg_{\frac{M}{BD}}\left(\frac{n}{BD}\right)\right)$ I/Os (same reasonament with I/O read BD items)

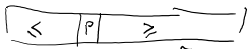
Col 2 Multi-way Mergesort is time and I/Os optimal if $D = 1$.

($D > 1$ see Disc striping)

Distribuition-based sorting

mercoledì 1 marzo 2023 16:20

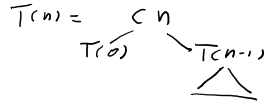
```
QUICKSORT(S,i,j)
  if (i<j){
      v← pick the position of "a good pivot";
      S← SWAP(S,v,i); // swap S[v] with S[i]
      p← PARTITION(S,i,j); // new position of the pivot
      QUICKSORT(S,i,p-1);
      QUICKSORT(S,p+1,j);
  }
```

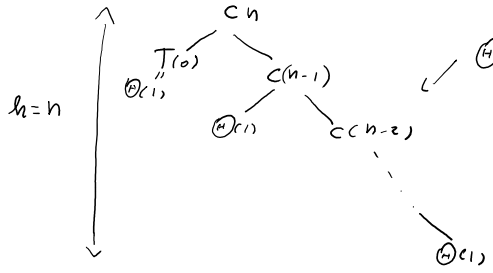


- Extra space: NO
- Time: $O(n \lg n)$ for balanced partitioning
  $O(n^2)$ for unbalanced "

Worst case (unbalanced): $T(n) = T(0) + T(n-1) + cn$ ← partition cost

recursion tree:





$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$

Total cost:

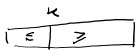
$T(n) = \sum_{i=1}^{n} \Theta(1) + \Theta(n^2)$

$= \Theta(n) + \Theta(n^2)$

$= \Theta(n^2)$

Best case: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \lg n)$ (see Merge Sort)

Average Case:



for $k = 0, \dots, n-1$ $\quad X_k := \begin{cases} 1 & \text{if PARTITION generates } k : n-k-1 \text{ split} \\ 0 & \text{otherwise} \end{cases}$

we assume $E[X_k] = 1 \cdot p(X_k = 1) + 0 \cdot p(X_k = 0) = p(X_k = 1) = \frac{1}{n}$ (all splits equally likely)

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split } (\text{if } X_0 = 1) \\ \vdots \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split } (\text{if } X_{n-1} = 1) \end{cases}$$

$$T(n) = \sum_{k=0}^{n-1} X_k \times (T(k) + T(n-k-1) + \Theta(n))$$

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \times (T(k) + T(n-k+1) + \Theta(n))\right]$$

$$= \sum_{k=0}^{n-1} E[X_k] \times E[T(k) + T(n-k+1) + \Theta(n)]$$

$$= \frac{1}{n} \sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n} \sum_{k=0}^{n-1} E[T(n-k+1)] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n)$$

$$= \frac{2}{n} \sum_{k=0}^{n-1} E[T(k)] + \Theta(n) = \frac{2}{n} \sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

$$= \frac{2}{n}\sum_{k=0}^{n-1} E[T(k)] + \Theta(n) \overset{k=0,1 \text{ absorbed by } \Theta(n)}{=} \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

Now, prove by induction that $E[T(n)] \le a n \lg_2 n$ for const $a > 0$

base case $n=2$: choose a large enough

$n \to n+1$: $E[T(n)] = \frac{2}{n}\sum_{k=1}^{n-1} E[T(k)] + \Theta(n) \overset{\text{induct. hp.}}{\le} \frac{2}{n}\sum_{k=2}^{n-1} ak \lg_2 k + \Theta(n)$ (we can choose the max a)

$\sum_{k=2}^{n-1} k \lg k = \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2$

$$\le \frac{2a}{n}\left(\frac{1}{2}n^2 \lg_2 n - \frac{1}{8}n^2\right) + \Theta(n)$$

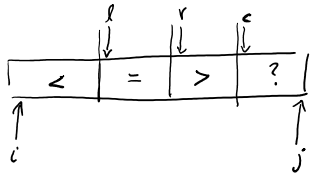
$$= an \lg_2 n - \left(\frac{an}{4} - \Theta(n)\right)$$

$$\le an \lg_2 n$$

if a chosen s.t. $\frac{an}{4}$ dominates $\Theta(n)$

```
THREE-WAY_PARTITIONING_QUICKSORT(S, i, j):
    if(j > i){
        (l, r) ← THREE-WAY_PARTITION(S, i, j);
        THREE-WAY_PARTITIONING_QUICKSORT(S, i, l-1);
        THREE-WAY_PARTITIONING_QUICKSORT(S, r, j)
    }
```

```
THREE-WAY_PARTITION(S, i, j)
    p ← S[i]; // pivot element
    l ← i;
    r ← i+1;
    for (c = r, ..., j){
        if (S[c] == p){
            SWAP(S[c], S[r]);
            r++;
        }
        else if (S[c] < p){
            CYCLE(S[l], S[r], S[c]);
            r++;
            l++;
        }
    }
```



- Extra space: NO
- Time: $O(n \log n)$ on avg. (*)

(*) Thm Random selection of the pivot => the algorithm computes

(*) **Thm** Random selection of the pivot $\Rightarrow$ the algorithm compares no more than $2n \ln n$ items on avg.

Proof Let be RV $X_{u,v} = \begin{cases} 1 & \text{if } S[u] \text{ and } S[v] \text{ are compared by PARTITION} \\ 0 & \text{otherwise} \end{cases}$

$$P_{u,v} = P(X_{u,v} = 1)$$

total # comparisons $\rightarrow E\left[\sum_{u,v}^{n} X_{u,v}\right] = \sum_{u}^{n} \sum_{v>u}^{n} E[X_{u,v}] = \sum_{u}^{n} \sum_{v>u}^{n} (1 \times P_{u,v} + 0 \times (1 - P_{u,v})) = \sum_{u}^{n} \sum_{v>u}^{n} P_{u,v}$

We have 3 cases:

1) pivot $< S[u], S[v]$ OR $S[u], S[v] <$ pivot } some RECURSIVE CALL (no utility for computation of $P_{u,v}$)

2) $S[u] <$ pivot $< S[v]$ OR $S[v] <$ pivot $< S[u]$ } different RECURSIVE CALL $\rightarrow u, v$ NOT compared

3) pivot $= S[u]$ OR pivot $= S[v]$ } $\Rightarrow u, v$ compared

Considering cases 2) and 3) and a bijective map $\phi: S \longrightarrow S' = \text{sorted}(S)$, $S[u] \longrightarrow S'[u']$

$$P_{u,v} = \frac{2}{v' - u' + 1} \quad \left(= \frac{\#\text{cases 2)}}{\#\text{cases 3)}} \right)$$

$$\Rightarrow E\left[\sum_{u,v}^{n} X_{u,v}\right] = \sum_{u=1}^{n} \sum_{v=u+1}^{n} P_{u,v} = \sum_{u'=1}^{n} \sum_{v'=u'+1}^{n} \frac{2}{v' - u' + 1}$$

$$= 2 \sum_{u'=1}^{n} \sum_{k=2}^{n-u'+1} \frac{1}{k} \leq 2 \sum_{u'=1}^{n} \sum_{k=2}^{n} \frac{1}{k} \leq 2n \ln n$$

$\sum_{k=1}^{n} \frac{1}{k} \leq 1 + \ln n$ ▨

```
RANDSELECT(S, k) // Select k-th ranked item
    r ← RAND(1, n);
    (S<, S>) ← PARTITION(S, r); // items respectively < and > than S[r]
    n< ← |S<|;
    n= ← |S| − (|S<| + |S>|);
    if (k ≤ n<)
        return RANDSELECT(S<, k);
    else if (k ≤ n< + n=)
        return S[r]; // S[r] is in S=
    else
        return RANDSELECT(S>, k − n< − n=); // skipping the items in S< ∪ S=
```

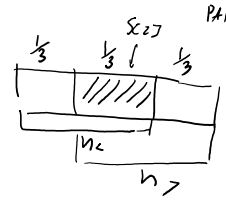
- Extra space: NO
- Time: $O(n)$ in avg.
- I/Os: $O(\frac{n}{B})$ } (*)

(*) **Thm** Selecting k-th ranked item in an unsorted sequence of size n takes $O(n)$ avg time in RAM model and $O(\frac{n}{B})$ I/Os in Disc model

Proof Let be $p_< =$ prob. of go to left
$p_> =$ " " " right

$$T(n) = O(n) + p_< T(n_<) + p_> T(n_>)$$
↑ PARTITION

good case: $n_<, n_> \le \frac{2}{3}$:



To ensure this S[?] must be in $[\frac{1}{3}n, \frac{2}{3}n]$
$\Rightarrow p = \frac{1}{3}$ (of a good case)

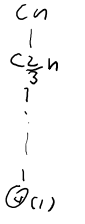
bad case: $n_<$ OR $n_> > \frac{2}{3}$

$\Rightarrow p = \frac{2}{3}$ (if a bad case)

$$\Rightarrow E[T(n)] \le O(n) + \frac{1}{3} \cdot E\left[T\left(\tfrac{2}{3}n\right)\right] + \frac{2}{3} \cdot E[T(n)]$$
↑ partition — GOOD CASE — BAD CASE — pessimistic analysis

$$\frac{1}{3}E[T(n)] \le O(n) + \frac{1}{3}E\left[T\left(\tfrac{2}{3}n\right)\right]$$

$$E[T(n)] \le O(n) + E\left[T\left(\tfrac{2}{3}n\right)\right] = O(n)$$
↖ recursive tree



cost
$= \left[\sum_{k=1}^{\log_{3/2} n} \left(\frac{2}{3}\right)^k\right] \cdot cn$
$\le \left[\sum_{k=1}^{+\infty} \left(\frac{2}{3}\right)^k\right] \cdot cn$
$= \frac{1}{1-\frac{2}{3}} \cdot cn = 3cn = O(n)$

on Disk model:

$$E[T(n)] \le O(n/B) + E\left[T\left(\tfrac{2}{3}n\right)\right] = O(n/B)$$
↑ scanning of $S_<, S_=, S_>$ in a single pass

□

Quick Sort is in place but we must consider the managment of the recursive calls

#nested recursive calls $= \begin{cases} \Theta(\lg n) & \text{best case} \\ \Theta(n) & \text{worst case} \end{cases}$ (For each rec. call $\Theta(1)$ space cost)

So $O(n)$ extra working space. Solution:

```
BOUNDEDQS(S, i, j):
  while(j - i > n0){
      r ← position of a good pivot;
      S ← SWAP(S, r, i);
      p ← PARTITION(S, i, j);
      if (p ≤ (i+j)/2){
          BOUNDEDQS(S, i, p-1);
          i = p+1;
      } else {
          BOUNDEDQS(S, p+1, j);
          j = p-1;
```

"elimination of tail recursion"

- Extra space: $O(\lg_2 n)$ (*)
- Time: $O(n \lg n)$ avg.

```
        BOUNDEDQS(S, p+1, j);
        j = p-1;
    }
}
INSERTION SORT (S, i, j);
```

(*) Thm On RAM model BOUNDEDQS uses $O(\lg_2 n)$ extra working space.

Proof The recursive call is executed on a subarray whose size is no more than half of the starting array. So # nested recursive call $= O(\lg_2 n)$ ☐

MULIT-WAY QUICKSORT

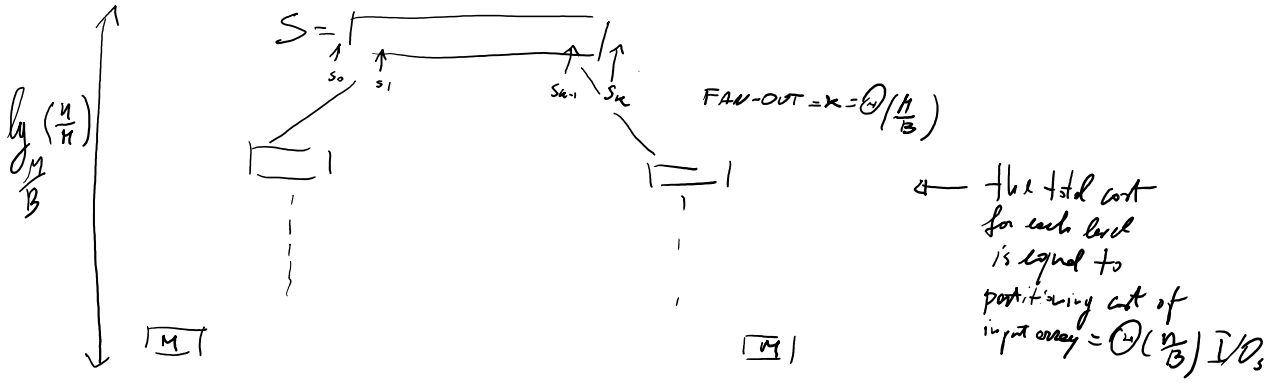
$k-1$ pivot $s_1, \dots, s_{k-1}$ s.t. $k = \Theta\left(\frac{M}{B}\right)$ $\quad s_0 = 0, s_k = +\infty$ dummy pivot

$B_i = \{S[j] \mid s_{i-1} \le S[j] < s_i\}$

We suppose good choice of pivots $\Rightarrow |B_i| = \Theta\left(\frac{n}{k}\right)$

The partition can be implemented in $O\left(\frac{n}{B}\right)$ using 1 input block (to read input seq.) and k output block (to write into the partition)

Using recursion tree:



We have $\Theta\left(\frac{n}{B}\right)\left(\Theta\log_{\frac{M}{B}} n - \Theta\log_{\frac{M}{B}} M\right) = \Theta\left(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{M}\right)$ I/Os (Optimal)

↑ partitioning ↑ Sequence inside M

GOOD SELECTION of $k-1$ PIVOTS VIA OVERSAMPLING ($a \ge 0$ int)

```
A ← SAMPLING(S, (a+1)k - 1);
A ← SORT(A);
```

- Time: $O(k \lg^2 k)$ (*) taking $a = \Theta(\lg k)$

```
A[-] ← SORT(A);
for i = 1, ---, k-1
        OUTPUT(s_i = A[(a+1) i]);
```

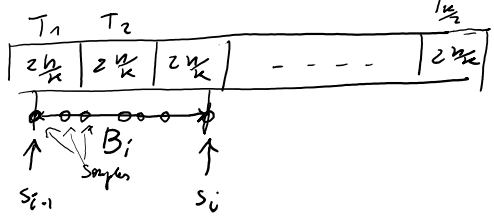
(*) Prop if $a = \Theta(\lg n)$ oversampling tone time $O(k \lg^2 k)$. Proof time sort $A = O(an \lg(an)) = O(\lg n \cdot n \cdot \lg(\lg n \cdot n))$
$= O(\lg n \cdot n \cdot [\lg(\lg n) + \lg k]) = O(n \cdot \lg^2 k)$ ∎

Lemma Let $k \geq 2$ and $a+1 = 12 \ln k \Rightarrow |B_i| < \frac{4n}{k}$ with probability $\geq \frac{1}{2}$

Proof By contradiction:

we suppose $P(\exists B_i \mid |B_i| \geq \frac{4n}{k}) > \frac{1}{2}$

$S' = \text{Sorted}(S) =$



$P(\exists B_i \mid |B_i| \geq \frac{4n}{k}) \leq P(\exists T_j \mid T_j \subset B_i) = P(\exists T_j \mid T_j \text{ contains} < (a+1) \text{ samples})$

$= \sum_{j=1}^{k/2} P(T_j \text{ contains} < (a+1) \text{ samples})$

$\leq \frac{k}{2} P(T_j \text{ contains} < (a+1) \text{ samples})$ (union bound) (*)

(*) $P(\text{a sample is in } T_j) = \frac{2n/k}{n} = \frac{2}{k}$ (uniform sampling) $= P(X_i)$ / let $X = \#\text{samples in } T_j$

$X_i = \begin{cases} 1 & \text{if } i\text{th sample is in } T_j \\ 0 & \text{otherwise} \end{cases}$

$E[X] = E[\sum_{i=1}^{\#\text{sample}} X_i] = \sum_{i=1}^{\#\text{sample}} E[X_i] = \sum_{i=1}^{\#\text{sample}} P(X_i = 1) = [(a+1)k - 1] \frac{2}{k}$

($(a+1)k - 1$: # sample; $\frac{2}{k}$: $P(\text{generic sample in } T_j)$)

So $E[X] = 2(a+1) - \frac{2}{k} \geq 2(a+1) - 1 \geq \frac{3}{2}(a+1)$

($k \geq 2$; $a \geq 1$)

So $a+1 \geq (1 - \frac{1}{3}) E[X]$ (*)

$\Rightarrow P(X < a+1) \underset{(*)}{\leq} P(X < (1 - \frac{1}{3}) E[X]) \underset{\text{Chernoff Bound}}{\leq} e^{-\frac{(1/3)^2}{2} \cdot E[X]} = e^{-\frac{E[X]}{18}} \underset{(*)}{\leq} e^{[-\frac{3}{2}(a+1)]/18}$

$= e^{-(a+1)/12} = e^{-\ln k} = \frac{1}{k}$

($a = 12 \ln k - 1$)

Finally:

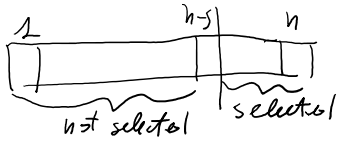
$P(\exists B_i \mid |B_i| \geq \frac{4n}{k}) \leq \frac{k}{2} \cdot \frac{1}{k} = \frac{1}{2}$

∎

Disk model (known seq.)

lunedì 13 febbraio 2023 20:11

$S[1,n] = s_1, \dots, s_n$ $(m < n)$ stored on disk

Aux $S'[1,n] =$ 


```
S'[1,n] ← S[1,n];
for s = 0,1,...,m {
    p ← RAND(1, n-s);
    SELECT(S'[p]);
    SWAP(S'[p], S'[n-s]);
}
```

- Extra space: $O(n \lg n)$ bits ← S' array of pointers
- Time:
- I/Os: $\Theta(m)$ I/Os ← access in cells in pure random way & swap items

Dictionary

```
Q ← ∅
while (|Q| < m) {
    p ← RAND(1, n);
    if p ∉ Q
        INSERT(p, Q);
}
```

Fact $\Pr(p \in Q) < \frac{1}{2}$ and $O(1)$ resamplings in avg.

Dim $\Pr(p \in Q) = \frac{|Q|}{n} \le \frac{m}{n} < \frac{1}{2}$ ↑ s.p.g. $m < \frac{n}{2}$

$E[\#\text{resampling}] < \sum_{\ell=1}^{\infty} \ell \left(\frac{1}{2}\right)^{\ell} \cdot \ell = 2$ ∎

- Extra space: $O(m)$
- Time: $O(m)$ in avg.
- I/Os: $O(\min\{m, \frac{n}{B}\})$ I/Os

← Using Hash table with chaining

Sorting

```
S ← ∅
while (|S| < m) {
    X ← RAND((1,n), m);
    X ← SORT(X);
    S ← REMOVEDUPLICATE(S);
}
```

- Extra space: $O(m)$
- Time: $O(m \lg m)$ avg.
- I/Os: $O(\min\{m, \frac{n}{B}\})$ I/Os if $m \le M$ small enough to keep sampled pos. in I.M.

Prop $P(\exists \text{ duplicate in } S) \le e^{-m(m-1)/2n}$

Dim Birthday problem:

$$P(\nexists \text{ duplicate in } S) = \frac{m!\binom{n}{m}}{n^m} = \frac{n(n-1)\cdots(n-m+1)}{n^m} = 1\times\left(\frac{n-1}{n}\right)\times\cdots\times\left(\frac{n-m+1}{n}\right)$$

$$= 1\times\left(1-\frac{1}{n}\right)\times\cdots\times\left(1-\frac{m-1}{n}\right) \le e^0\times e^{-1/n}\times\cdots\times e^{-\frac{(m-1)}{n}} = e^{-\left(\sum_{i=1}^{m-1} i\right)/n}$$

$$\left(e^x \ge 1+x\right) \quad = e^{-\frac{m(m-1)}{2n}}$$

∎

Corollary If $m = \sqrt{n}$ $P(\text{duplicate in } S) \gtrsim 1 - \frac{1}{\sqrt{e}} \approx 0.4$ $\left(= P(\text{re-sampling})\right)$

Dim $P(\nexists \text{ dup. in } S) \le e^{-\frac{1}{2}} = \frac{1}{\sqrt{e}}$ ∎

Streaming model (known seq.)

lunedì 13 febbraio 2023 21:57

$S[1,n] = s_1, \dots, s_n$, $m < n$

SCANNING & SELECTING (S, m):

```
S ← 0 // # selected
for (j = 1; j ≤ n & S < m; j++) {
    p ← RAND(0,1);
    if p < (m-S)/(n-j+1) {
        SELECT(S[j]);
        S++;
    }
}
```

$p = \text{rand}(0,1)$ → NO → 1; YES → $\frac{m-S}{n-j+1} = p(\text{pick } s_j)$; 0

- Extra space: $O(m)$
- Time: $O(n)$
- I/Os: $O\left(\frac{n}{B}\right)$ I/Os

CASE $m = 1$: $p(\text{pick } s_j) = \frac{1}{n-j+1} \Rightarrow p(\text{sample } s_j) = \frac{1}{n}$

Proof By induction:

Suppose $p(\text{sample } s_i, i \in \{1, \dots, j-1\}) = \frac{1}{n}$

$$p(\text{sample } s_j, j \in \{1, \dots, n\}) = p(\text{not sample } s_1, \dots, s_{j-1}) \times p(\text{pick } s_j) =$$
$$= 1 - p(\text{sample } s_1 \text{ or } \dots \text{ or } s_{j-1}) \times p(\text{pick } s_j)$$
$$= \left(1 - \frac{j-1}{n}\right) \times \frac{1}{n-j+1} = \frac{1}{n}$$

CASE $m \geq 1$: $P(\text{pick } s_j) = \frac{m-S}{n-j+1}$

Proof: $p(\text{pick } s_j) = \frac{\#\text{ elem. to pick}}{\#\text{ remain. pos. to scan}} = \frac{|\{S' \subseteq S[j,n], |S'| = m-S, s_j \in S'\}|}{|\{S \subseteq S[j,n], |S'| = m-S\}|}$

$$= \frac{\binom{n-j}{m-S-1}}{\binom{n-j+1}{m-S}} = \frac{\cancel{(n-j)!}}{\cancel{(m-S-1)!}\cancel{(n-j-m+S+1)!}} \cdot \frac{\cancel{(m-S)!}\cancel{(n-j+1-m+S)!}}{\cancel{(n-j+1)!}} = \frac{m-S}{n-j+1}$$

Streaming model (unknown seq.)

lunedì 27 febbraio 2023 17:29

$S = s_1, \dots, s_n, \dots$

min-Heap $H = \{\langle r, item \rangle\}$ (r: random $\in [0,1]$; item $\in S$)

START: $H = \{\langle -\infty, \phi \rangle, \dots, \langle -\infty, \phi \rangle\}$, $|H| = m$

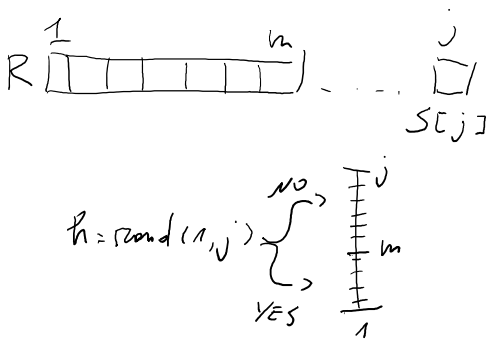
```
for j = 1, ..., n {
    r_j ← RAND(0,1);
    y ← MIN(H);
    if (r_j > y) {
        POP_MIN(H);
        INSERT(<r_j, S[j]>, H);
    }
}
return H;
```

- Extra space: $O(m)$
- Time: $O(n \lg m)$ ($O(\lg m)$ to insert item in H)
- I/Os: $O(n/B)$ I/Os

START: $R[1, m] = S[1, m]$

RESERVOIR SAMPLING:

```
for j = m+1, ..., n, ... {
    h ← RAND(1, j); // Rand integer
    if (h ≤ m)
        R[h] ← S[j];
}
return R;
```



- Extra space: (H) $O(m)$
- Time: $O(n)$
- I/Os: $O(n/B)$ I/Os

} optimal

Prop $p(s_j \in R \text{ after } n \text{ steps}) = \frac{m}{n}$ $(\forall 1 \le j \le n)$

Proof Case $n = m$: $p(s_j \in R) = 1 = \frac{m}{n}$ (start: $R[1, m] = S[1, m]$)

Case $n > m$: $p(s_j \in R \text{ after } n \text{ steps}) =$ ← $1 \le j \le n-1$ (s_n go inside R at n-th step with prob. $= \frac{m}{n}$)

$p(s_j \in R \text{ after } n-1 \text{ steps}) \times [p(s_n \text{ not picked}) + p(s_n \text{ picked}) \times p(s_j \text{ not removed})] =$

$$\frac{m}{n-1} \times \left[\left(1 - \frac{m}{n}\right) + \frac{m}{n} \times \frac{m-1}{m}\right] =$$

($\frac{m}{n-1}$: inductive step; $p(s_n \text{ picked}) = \frac{m}{n}$; $\frac{m-1}{m}$: $p(\text{remove } s_i \ne s_j \text{ from } R)$)

$= \sum_{i \ne j} p(\text{remove } s_i \text{ from } R) = \sum_{i \ne j} \frac{1}{m} = \frac{m-1}{m}$

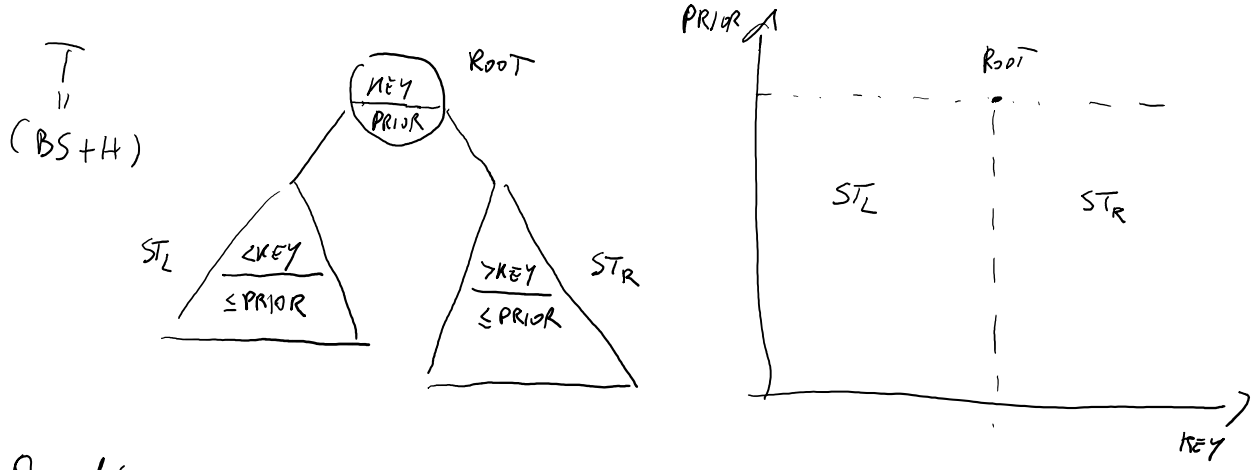
$$= \frac{m}{n-1} \times \left[1 + \left(\frac{m-1}{m} - 1 \right) \times \frac{n}{n} \right] = \frac{m}{n-1} \times \left[1 - \frac{1}{n} \right] = \frac{m}{n-1} \times \frac{n-1}{n} = \frac{m}{n}$$

Treap

lunedì 13 febbraio 2023 20:09

Def Treap T is a binary tree based data structure. Each node $u \in T$ is given by a search key and a priority $u = (\text{search key}, \text{priority})$.
The treap is simultaneously a binary search tree for the search key, and a max or min heap for the priority of the node.

$Node = (\text{search key}, \text{priority})$
↑ BIN. SEARCH TREE ↑ MIN/MAX HEAP

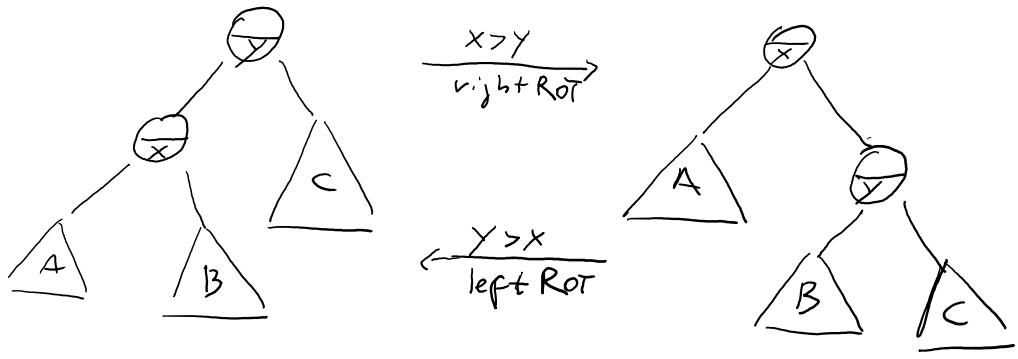


Operations

- Search(KEY=k)
- Insert(node=z)
- Delete(node=z)
- Split(T, KEY=k) = return 2 treap $T_<, T_>$ such that each key($T_<$) < key($T_>$)
- Merge($T_<, T_>$) = return a treap $T = T_< \cup T_>$ (key($T_<$) < key($T_>$))

```
SEARCH(T, key=k)
    return SEARCH(BST(T), k); // search k as in bin. search tree
```

```
INSERT(T, node=z)
    INSERT(BST(T), key(node));
    while (¬HEAP(T)){ // the priorities doesn't satisfy the heap
        ROTATION(T, z);
    }
    return T;
```




```
DELETE(T, node = z)
    while (z ≠ LEAF(T)) {
        ROTATION(T, z); // rotate with children with smaller priority (if minheap)
    }
    CUT(T, z);
    return T;
```

- Time = $O(n)$ worst case (*)$_1$
  $O(\lg_2 n)$ expected case (*)$_2$

```
SPLIT(T, key = k)                    min-heap  max-heap
    INSERT(T, node = (k, -∞/+∞)); // return a Treap with root = (k, ±∞)
    T< ← LEFT_SUBTREAP(T, root);
    T> ← RIGHT_SUBTREAP(T, root);
    return (T<, T>);
```

```
MERGE(T<, T>)                        max-heap  min-heap
    dummynode ← (?, -∞/+∞);
    T ← TREAP(ROOT = dummynode
              LEFT_SUBTREAP = T<
              RIGHT_SUBTREAP = T>);
    while (dummynode ≠ LEAF(T))
        ROTATION(T, dummynode);
    CUT(T, dummynode);
    return T;
```

Thm (*)$_1$ All the operation on treap time $O(n)$ time

Prof the time need by the operation is Θ(depth(node)) where node is the node that must be inserted, deleted, searched or used by split or merge.

Θ(depth(node)) = $O(h)$ = $O(n)$ (the tree is not necessary balanced)

node = leaf; worst case

□

node = leaf worst case

Random priorities

$PRIOR_i = PRIOR(node_i)$ $PRIOR_1, \dots, PRIOR_n$ i.i.d. RV, $PRIOR_i = random(0,1)$

Thm (*) Assuming min-treap, x_k generic node of T. $[E depth(x_k)] = O(\lg n)$

Proof x_k s.t. $x_k \in NODES(T) = \{x_1, \dots, x_n\}$

$KEY(x_1) < \dots < KEY(x_k) < \dots < KEY(x_n)$

↑ k-th smallest

- I.V. $A_k^i := \begin{cases} 1 & \text{if } x_i \text{ proper ancestor of } x_k \\ 0 & \text{otherwise} \end{cases}$

$depth(x_k) = \sum_{i=1}^{n} A_k^i$, $E[depth(x_k)] = \sum_{i=1}^{n} P(A_k^i = 1)$

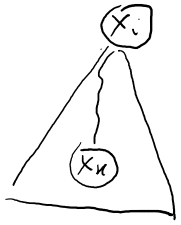
↑ A_k^i is IV

- $\forall i \neq k$ $X_{(i,k)} = X_{(k,i)} := \begin{cases} \{x_i, \dots, x_k\} & \text{if } i < k \\ \{x_k, \dots, x_i\} & \text{if } i > k \end{cases}$

Lemma Now we prove that: $A_k^i = 1 \iff PRIOR(x_i) = \min\{PRIOR(X_{(i,k)})\}$

→ we have 4 possible case of treap

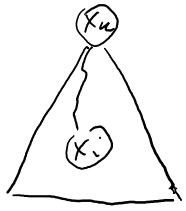
1) x_i is root



$PRIOR(x_i) = \min PRIOR(X_{(i,k)})$

$A_k^i = 1$ ok

2) x_k is root

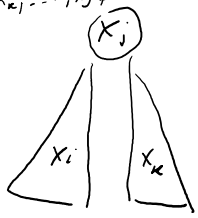


$PRIOR(x_i) \neq \min PRIOR(X_{(i,k)})$ $(= PRIOR(x_k))$

$A_k^i = 0$ ok

3) x_j root

$x_i, \dots, x_j, \dots, x_k$
OR $x_k, \dots, x_j, \dots, x_i$



OR

$PRIOR(x_i) \neq \min PRIOR(X_{(i,k)})$ $(= PRIOR(x_j))$

$A_k^i = 0$ ok

4) x_j root

OR

Recursively on the subtree

$(= PRIOR(x_j))$

$A^i_x = 0$ ok

$PRIOR_1, \dots, PRIOR_x$ i.i.d.

$$\Rightarrow p\left(PRIOR_i = \min\left(PRIOR(X_{(i,k)})\right)\right) \overset{(*)}{=} \frac{1}{|k-i|+1}$$ uniform distribution

And

$$p(A^i_k = 1) \overset{\text{Lemma}}{=} [i \neq n] \cdot p\left(PRIOR_i = \min(PRIOR(X_{(i,k)}))\right) =$$

$$= [i \neq n] \cdot \frac{1}{|k-i|+1} = \begin{cases} \frac{1}{k-i+1} & \text{if } i < k \\ 0 & \text{if } i = k \\ \frac{1}{i-k+1} & \text{if } i > k \end{cases}$$

$(*)$

Therefore,

$$E[\text{depth}(x_n)] = \sum_{i=1}^{n} p(A^i_n = 1) = \sum_{i<k} \frac{1}{k-i+1} + \sum_{i>k} \frac{1}{i-k+1}$$

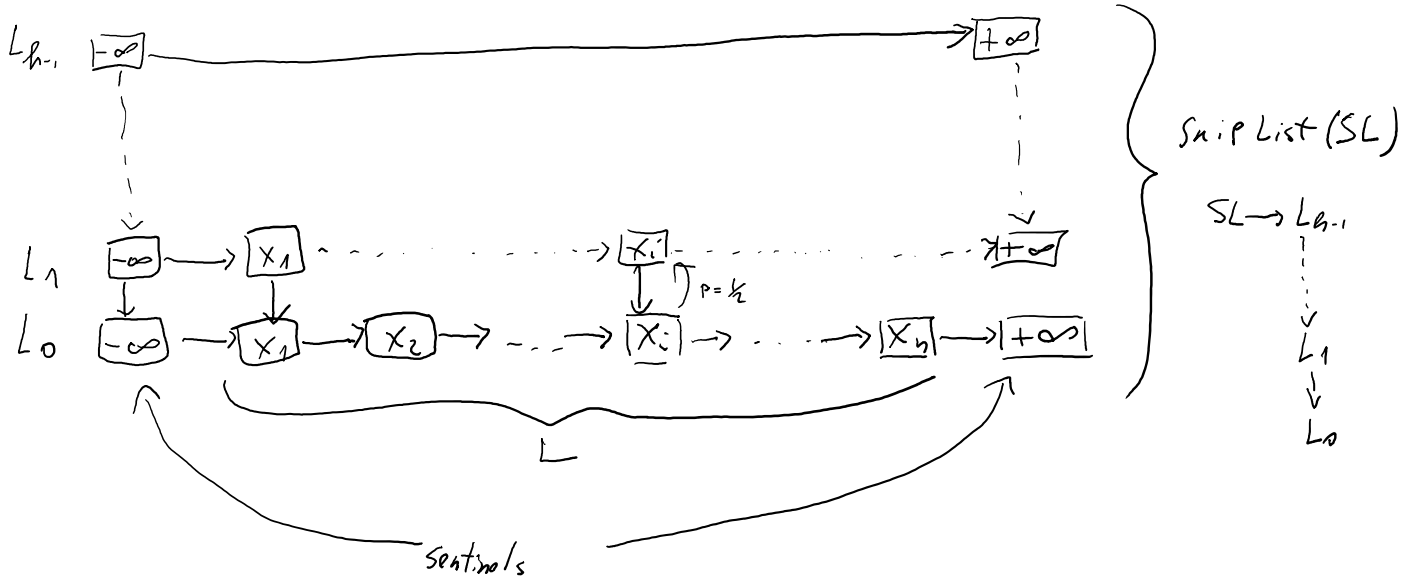
$$= \sum_{j=2}^{k} \frac{1}{j} + \sum_{j=2}^{n-k+1} \frac{1}{j} = \sum_{j=1}^{k} \frac{1}{j} - 1 + \sum_{j=1}^{n-k+1} \frac{1}{j} - 1 < \ln(k) + \ln(n-k+1) - 2$$

$$< 2\ln(n) - 2 = O(\lg n)$$

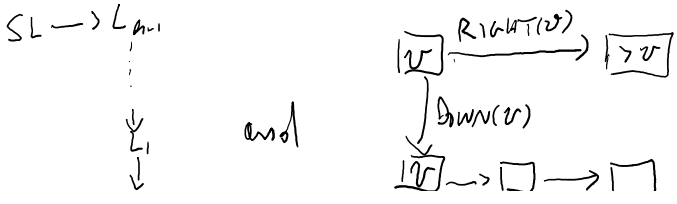
Skip List

giovedì 23 marzo 2023 00:54

List $L = x_1 < \dots < x_n$




```
BUILD_SKIPLIST(L)
    L0 ← L;
    L0 ← ADD_SENTINELS(L0); // add -∞ and +∞
    SL ← L0; // initialize Skip List
    TO_COPY ← ITEM(L0); // L0 without sentinels
    while (TO_COPY ≠ φ) {
        Li ← φ;
        for each ITEM in TO_COPY {
            p ← RAND(0,1);
            if p < 0.5
                ADD(DUPLICATE(ITEM), Li);
        }
        Li ← ADD_SENTINELS(Li);
        SL ← ADD(SL, Li);
        TO_COPY ← ITEM(Li);
        i++;
    }
```




```
SEARCH(k, SL)
    v ← SL; //point first item of L_h
    while (v ≠ NULL AND KEY(v) ≠ k) {
        if KEY(RIGHT(v)) < k
            v ← RIGHT(v);
        else
            v ← DOWN(v);
    }
    return v;
```

• Time: $O(\log n)$ with high prob. ($*$)

Analysis

Def 1 $h :=$ levels of SL

Def 2 Given $-\infty < x < +\infty$, $C(x) :=$ #occ. of x in $L_1 \cup \dots \cup L_{h-1}$ (not L_0)
$=$ #time x is copied

Lemma $\forall$ key x in SL $\quad E[C(x)] = 1$

Proof Recursively $E[C(x)] = p(\text{not copy } x) \cdot \underbrace{0}_{C(x)=0} + p(\text{copy } x) \cdot \underbrace{[E[C(x)]+1]}_{C(x)+1}$

$\Leftrightarrow E[C(x)] = \frac{1}{2} \cdot 0 + \frac{1}{2} E[C(x)] + \frac{1}{2}$

$\Leftrightarrow E[C(x)] = 1$

Or equivalently $C(x)$ = geometric distribution $\quad C(x) = \#\text{failures (copy) before success (not copy)}$

$p = p(\text{not copy } x)$

$1 - p = p(\text{copy } x)$

$$E[C(x)] = \sum_{k=0}^{+\infty} k \cdot \underbrace{p(C(x)=k)}_{(1-p)^k \cdot p} = \sum_{k=0}^{+\infty} k \cdot (1-p)^k \cdot p$$

$$= \frac{1-p}{p} = \frac{1-\frac{1}{2}}{\frac{1}{2}} = 1$$

(see Geo. Distr.)

∎

Thm $p(h \geq c \cdot \log n) \leq \frac{1}{\dots}$ $\quad (h = O(\log n)$ with high prob)

Thm $p(h \geq c \cdot \lg n) \leq \frac{1}{n^{c-1}}$ ($h = O(\lg n)$ with high prob)

Proof $p(h \geq \ell) = p(C(x_1) \geq \ell \vee \dots \vee C(x_n) \geq \ell)$

$\leq \sum_{i=1}^{n} p(C(x_i) \geq \ell) = n \cdot p(C(x) \geq \ell) \leq n \cdot \frac{1}{2^\ell}$

(Union Bound; i.i.d; at least ℓ copy)

So $\underbrace{p(h \geq \ell) \leq \frac{n}{2^\ell}}_{(*)} \Rightarrow p(h \geq c \lg_2 n) \leq \frac{n}{2^{c \lg_2 n}} \leq \frac{n}{n^c} = \frac{1}{n^{c-1}}$

Thm $E[h] \leq \lg n + 1$ (in avg. at most 1 levels more than ideal)

Proof $E[h] = \sum_{\ell=0}^{+\infty} \ell \, p(h = \ell) = \sum_{\ell \geq 1} p(h \geq \ell)$

using IV $I = \begin{cases} 1 & \text{if } h \geq p \\ 0 & \text{otherwise} \end{cases}$

$= \sum_{\ell=1}^{\lg n} p(h \geq \ell) + \sum_{\ell=\lg n+1}^{+\infty} p(h \geq \ell)$

$\leq \underbrace{\lg n \cdot \frac{n}{2^{\lg n}}}_{\text{using } (*)} + \underbrace{\sum_{i \geq 1} \frac{n}{2^{i+n}}}_{\text{substitute } \ell = i + \lg n} = \lg n + \sum_{i \geq 1} \frac{1}{2^i} = \lg n + \left(\frac{1}{1-\frac{1}{2}} - 1\right) = \lg n + 1$

Thm (*) Search every in skip list (copying with prob. $\frac{1}{2}$) take $O(\lg n)$ with high probability

Proof Search take $\Theta(h)$ time. $\Theta(h)$ can be bigger also more than $\Theta(n)$

But by thm before h is $O(\lg n)$ with high prob. $\Rightarrow O(\lg n)$ with high probability.

General

lunedì 13 febbraio 2023 20:04

Inverted list

lexicographically sorted

$w_1 \longrightarrow d[w_1] = d_1^{w_1}, \dots, d_{k_1}^{w_1}$

$\vdots$

$w_n \longrightarrow d[w_n] = d_1^{w_n}, \dots, d_{k_n}^{w_n}$

$d_j^{w_i}$ = j-th document of word w_i $(w_i \in d_j^{w_i})$

Query(w_1, w_2) ⟶ Intersect$(d[w_1], d[w_2])$

A ← set, B ← set

We suppose A and B sorted

if not sorted intersection time = $\begin{cases} O(m \times n) \\ O(m \lg m + n \lg n + \dots) \end{cases}$

↑ sort(A) ↑ sort(B) ↑ intersect(A,B)

$A = a_1, a_2, \dots, a_n \qquad a_i < a_{i+1}$

$B = b_1, b_2, \dots, b_m \qquad b_i < b_{i+1}$

Case $m \sim n$

```
MERGE-BASED INTERSECTION (A, B):
    i ← 1; j ← i
    a ← A[i]; b ← B[j];
    while (i<n OR j<m){
        if (a = b){
            OUTPUT(a);
            if(i<n) a ← A[i+1];
            if(j<m) b ← B[j+1];
        }
        else if (a<b AND i<n)
            a ← A[i+1];
        else if(j<m)
            b ← B[j+1];
```

- Extra space: NO
- Time: $O(m+n)$ Optimal if $n = \Theta(m)$, $(O(n))$
- I/Os: $O(\frac{m}{B} + \frac{n}{B})$ Optimal if $n = \Theta(m)$ $(O(\frac{n}{B}))$
- (+) cache friendly
- (-) bad if $m \ll n$
- (+) optimal if $m = n$

Case $m \ll n$

```
BINARY SEARCH INTERSECTION (A, B)
    for i = 1, ..., m
        BINARYSEARCH(A, B[i]); // binary search b_i in A
```

- Extra space: O
- Time: $O(m \lg n)$
- I/Os: $O(\frac{m}{B} \lg_2 \frac{n}{B})$
- (-) Not cache friendly
- (+) optimal if $m \ll n$

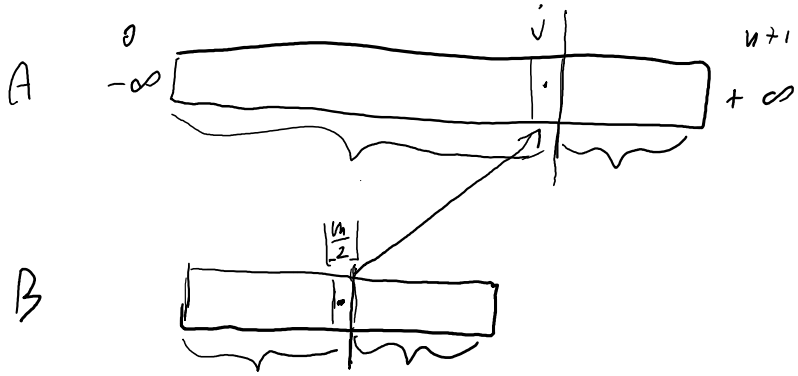
Lower Bound (Comparison-based)

Supposing $B \subseteq A$ $(m \leq n)$

$$\Omega\left(\lg \binom{n}{m}\right) = \Omega\left(m \lg \left(\frac{n}{m}\right)\right)$$

Mutual partitioning

lunedì 6 marzo 2023 11:52




```
MUTUAL PARTITIONING(A, B)
    n ← LENGHT(A); m ← LENGHT(B);
    p ← B[⌊m/2⌋]; // if n<m exchange the role of A and B
    i ← BINARYSEARCH(p, A); // a_i < b_⌊m/2⌋ < a_{i+1}
    if p = A[i]
            OUTPUT(p);
    MUTUAL PARTITIONING(A[1,i], B[1,⌊m/2⌋-1]); // a_i can be discarded if just outputted
     ''      ''          (A[i+1,n], B[⌊m/2⌋+1,m]);
```

- Extra space: NO
- Time: $O(m(1+\log \frac{n}{m}))$
(+) Combine the cost of the merge alg. and bin. search alg.
(-) NO cache friendly

fact time is $O(m(1+\log \frac{n}{m}))$

Proof $T(n,m) = O(\log n) + T(n_1, \frac{m}{2}) + T(n_2, \frac{m}{2}) \leq O(\log n) + 2T(\frac{n}{2}, \frac{m}{2})$

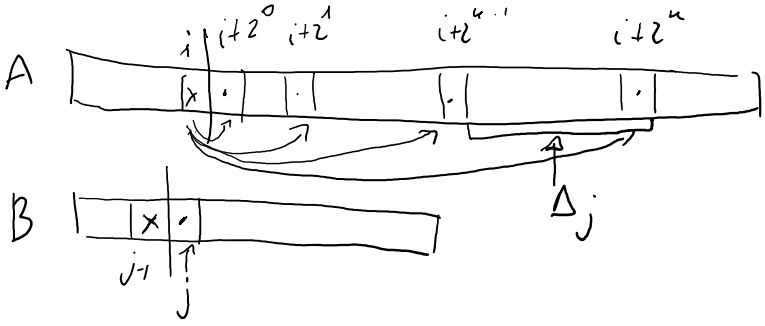
($O(\log n)$: BS; $\leq$: balanced case (worst case))

and $T(n,m) = 1$ if $n, m \leq 1$

$\Rightarrow T(n,m) = O(m(1+\log \frac{n}{m}))$ for any $m \leq n$ (not proved)

Doubling searching

lunedì 6 marzo 2023 12:37



DOUBLING SEARCH INTERSECTION(A, B)

```
m ← |B|; n ← |A|   // m ≤ n otherwise exchange A with B
i ← 0;
for j = 1, ..., m {
    k ← 0;
    while (i + 2^k ≤ n AND B[j] > A[i + 2^k])
        k ← k + 1;
    i' ← BINARYSEARCH(B[j], A[i + 2^{k-1} + 1, min{i + 2^k, n}]);  // BS in A[Δ_j]
    if (A[i'] = B[j])
        OUTPUT(B[j]);
    i ← i';
}
```

- Extra space: NO
- Time: $O(m(1 + \lg \frac{n}{m}))$ (*)

(+) Optimal in Comparison model
(+) NO recursion
(−) jumping

(*) **Fact** Time is $O(m(1 + \lg \frac{n}{m}))$. Optimal for comparison model.

Proof Let i_j the position of b_j in A (for BS).

$$A[i_{j-1} + 2^{k-1}] < b_j \leq A[i_{j-1} + 2^k] \Rightarrow 2^{k-1} < i_j - i_{j-1} \leq 2^k \quad (1)$$

(2) $\Delta_j = \min\{2^{k-1}, n\}$ (size of sub-array of A for BS).

$$\Rightarrow \Delta_j \overset{(2)}{\leq} 2^{k-1} \overset{(1)}{<} i_j - i_{j-1} \Rightarrow \Delta_j < i_j - i_{j-1} \text{ and } \sum_{j=1}^{m} \Delta_j < \sum_{j=1}^{m} (i_j - i_{j-1}) = i_m - i_0 \leq n$$

(telescopic sum)

$$T(n) = O\left(\sum_{j=1}^{m} (1 + \lg \Delta_j)\right) = O\left(m + \sum_{j=1}^{m} (\lg \Delta_j)\right) = O\left(m\left(1 + \frac{1}{m}\sum_{j=1}^{m} (\lg \Delta_j)\right)\right) =$$

(while statement; BS)

$$\overset{(*)}{=} O\left(m\left(1 + \lg\left(\sum_{j=1}^{m} \frac{\Delta_j}{m}\right)\right)\right) =$$

while statement (5)

$$\xrightarrow{(*)} = O\left(m\left(1+\lg\left(\frac{\sum_{j=1}^{m} \frac{A_j}{m}}{m}\right)\right)\right) =$$

$$= O\left(m\left(1+\lg\frac{n}{m}\right)\right)$$

(*) Jensen inequality:

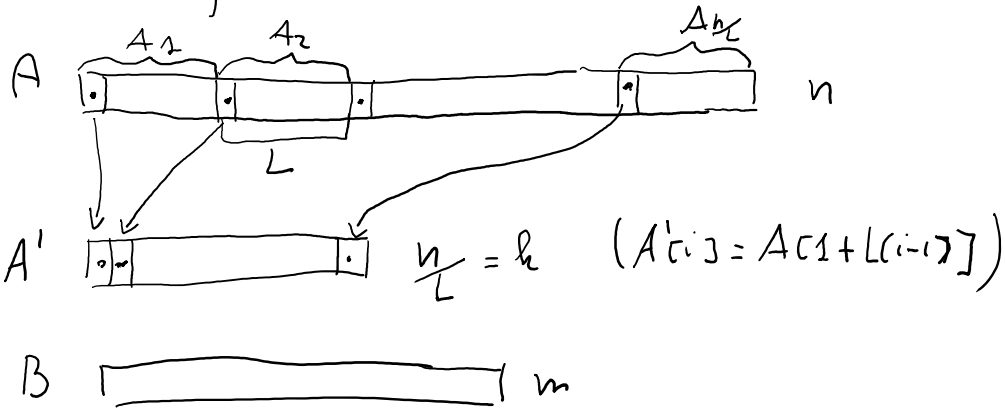
$$\begin{cases} \varphi\left(\sum_{j=1}^{m} \frac{x_j}{m}\right) \le \sum_{j=1}^{m} \frac{\varphi(x_j)}{m} & \text{if } \varphi \text{ is convex} \\ \varphi\left(\sum_{j=1}^{m} \frac{x_j}{m}\right) \ge \sum_{j=1}^{m} \frac{\varphi(x_j)}{m} & \text{if } \varphi \text{ is concave (*)} \end{cases}$$

also for all convex linear combination of $x_1, \dots, x_m$

2-level storage approach

lunedì 6 marzo 2023 16:39

L = size of block



2LEVEL INTERSECTION(A, A', B)

$(B_1, \dots, B_h) \leftarrow$ MERGE(A', B); // $B_i = \{B[j] \mid A'[i] \le B[j] < A'[i+1]\}$

for $i = 1, \dots, h$ {
  if $B_i \neq \phi$
    MERGE BASED INTERSECTION(A_i, B_i);
}

- Extra space: $O(\frac{n}{L})$ (A')
- Time: $O(\frac{n}{L} + mL)$
- I/Os: $O(\frac{n}{LB} + \frac{mL}{B} + m)$ } (*)

(*) Fact Time is $O(\frac{n}{L} + mL)$ and #I/Os are $O(\frac{n}{LB} + \frac{mL}{B} + m)$ I/Os

Proof

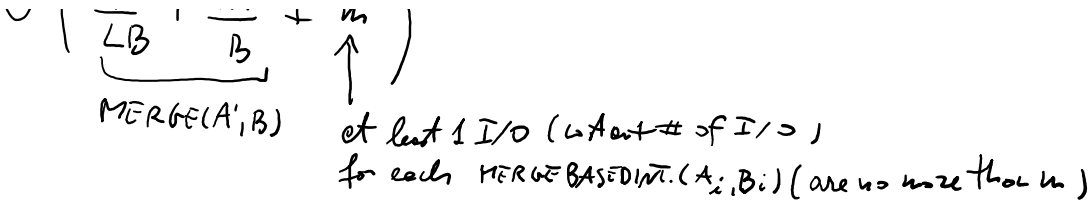
$$T(n) = \underbrace{O(\frac{n}{L} + m)}_{\text{MERGE}(A', B)} + \sum_{B_i \neq \phi} O(\underbrace{|A_i| + |B_i|}_{\text{MERGE BASED INT.}(A_i, B_i)}) = O(\frac{n}{L} + m) + O(\sum_{B_i \neq \phi} |A_i| + \sum_{B_i \neq \phi} |B_i|)$$

$$\overset{(*)}{=} O(\frac{n}{L} + m + mL + m)$$

(↑ at most $|B_i|$ touch m distinct A_i)

$$= O(\frac{n}{L} + mL)$$

I/Os: $O(\underbrace{\frac{n}{LB} + \frac{m}{B}}_{\text{MERGE}(A', B)} + \underbrace{m}_{\text{MERGE BASED INT.}(A_i, B_i)})$

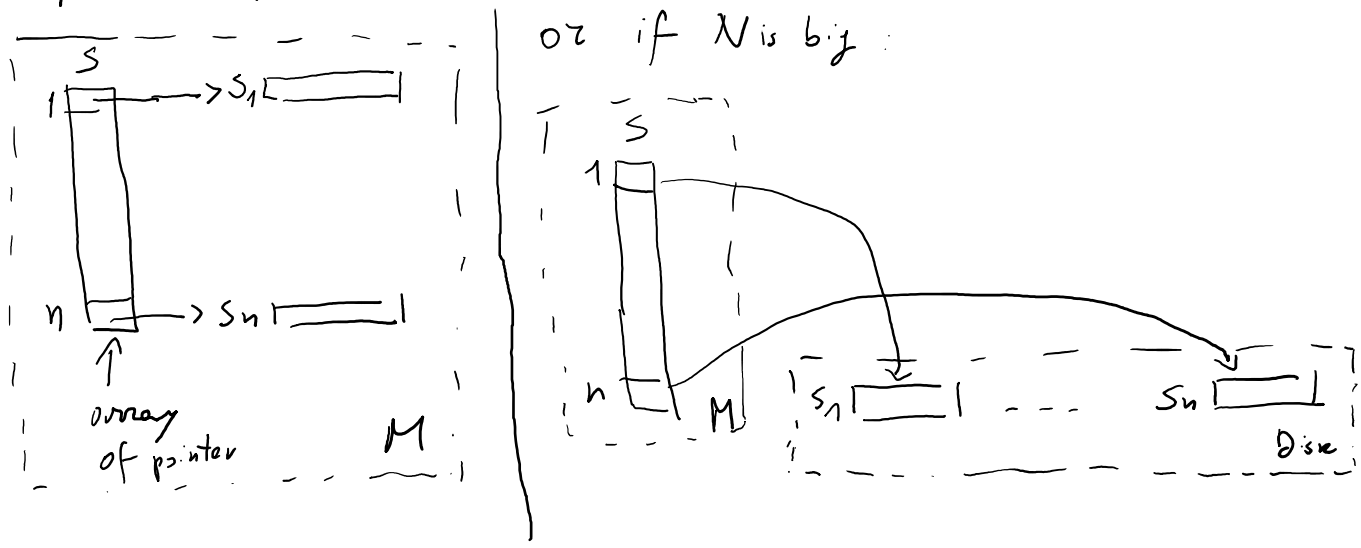

MERGE(A',B)
at least 1 I/O (wAmount of I/O)
for each MERGEBASEDINT.(Ai,Bi) (are no more than m)

$S = s_1, \dots, s_n$ $\quad n = \#\text{strings}$ $\quad \Sigma$ alphabet, $\sigma = |\Sigma|$

$N = \text{total length} = \sum_{i=1}^{n} |s_i|$

$L = \text{avg. length} = \frac{N}{n}$

Comparison based SORTER:



- $O(L n \lg n) = O(N \lg n)$ time
  - $O(L)$ for each string comparison
  - Comp. based sorter for n items

- Every time comparing two string s_i and s_j we must access to the string by the two pointer $\Rightarrow \Omega(n \lg n)$ I/Os

Lower Bound

Lemma $d_S \doteq$ shortest prefix of s_i that distinguish it from others.

The lower bound of the string sorting problem is $\Omega(d + n \lg n)$ comparison, where $d \doteq \sum_{s \in S} d_s$ i the distinguishing prefix of S

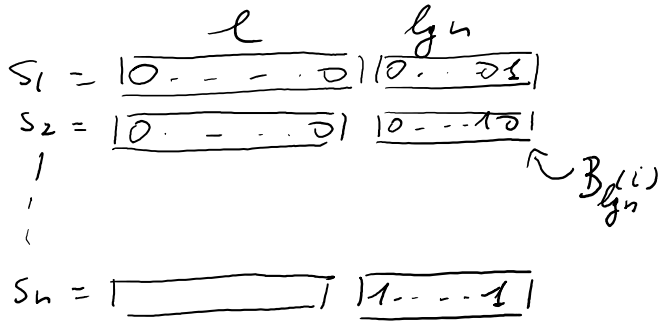
Patological case

Pathological case

$s_1, \dots, s_n$ stored in bit

$|s_i| = \ell + \log n$, $d_s = \ell + \log n$

$$d = \sum_{s \in S} d_s = n(\ell + \log n)$$



Lower bound $= \Omega(d + n \log n) = \Omega(n(\ell + \log n) + n \log n) = \Omega(N + n \log n) = \Omega(N)$

QuicksortMergeSort $= O(n(\ell + \log n) \cdot \log n) = O(N \log n)$ differs $\log n$ by optimal

Radix sort (MSD-first)

martedì 7 marzo 2023 15:12

String as a sequence of characters $s \in S$, $s = a_1 \dots a_{|s|}$, $a_i \in \{0, 1, \dots, \sigma-1\}$

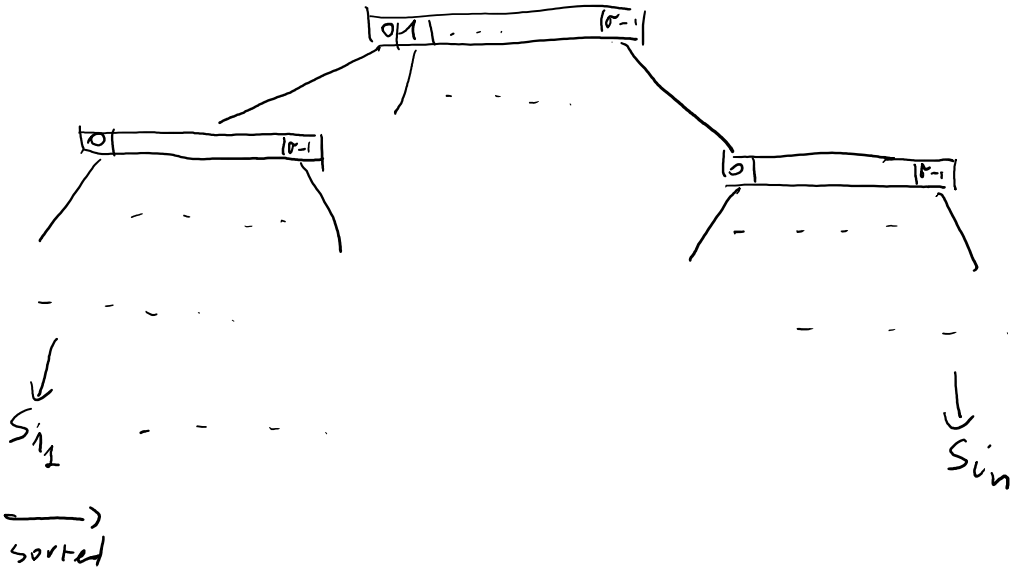
Naming process (assuming just given) $O(N \lg \sigma)$

Space for S' $\Theta(N \lg \sigma)$ bits (N = # chars, $\lceil \lg \sigma \rceil$ bits for each character)

MSD-first

Distribution-based approach → σ-ary tree (trie)

$S' = \{s_1, \dots, s_n\}$



- Extra space: $O(\#\text{internal node} \cdot \text{size(node)})$

internal node = #branching node + # unary node = $O(n) + O(N) = O(N)$

Optimization: Drop unary nodes without branching node as descendant node

$\Rightarrow$ # internal node = $O(d)$

```
MSD-FIRST_ARRAY(S):
    trie ← φ; // Empty trie
    for i = 1, ..., n
        trie ← INSERT(S_i, trie);
    PREORDERVISIT(trie);
```

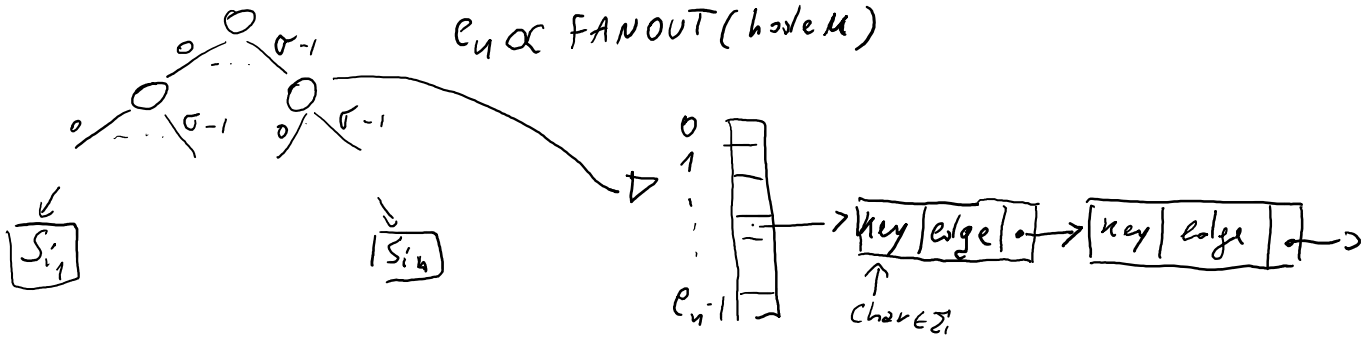
- Extra space: $O(d\sigma)$ (d: #nodes, σ: size(node))
- Time: $O(d\sigma)$

(*)

(*) **Thm** Extra space and time are $O(d\sigma)$ (using array)

Proof To create a new internal node in the trie we require $O(\sigma)$ time and extra space (allocating the array). The number of internal nodes are $O(d) \Rightarrow O(d\sigma)$ ∎

Using hash table (with chaining) instead of array:




```
MSD-FIRST_HASHTABLE(S):
    // as MSD-FIRST_ARRAY(S)
```

- Extra space: $O(d)$
- Time: $O(d \log \sigma)$ in avg.

(*)

(*) **Thm** Using hash with chaining time is $O(d \log \sigma)$ in avg and extra space is $O(d)$.

Proof time in avg = time (① construct trie) + time (② sort edges) + time (③ pre-order visit)

① = time (insert all strings) ≤ #node · time (access the node)

① = time(insert all strings) ≤ #node · time(access the node)
≤ d · O(1) in exp.

time(access the node) ≤ exp. lenght of chaining = $O\left(\frac{\#keys}{c_u}\right) = O\left(\frac{\sim\sigma}{\sim\sigma}\right) = O(1)$

② = time(sort pointers of hash tables) = $O\left(\sum_u c_u \cdot \log c_u\right) = O\left(\sum_u c_u \log \sigma\right) = O(d \log \sigma)$

$\sum_u$: sum over all nodes; c_u: #pointers; $\sum_u c_u \propto$ #internal node

③ = time(scan trie rightward) = time(access node) · #node ≤ $O(1) \cdot d = O(d)$

⇒ time = ① + ② + ③ = $O(d \log \sigma)$ in exp

Extra space ≤ space(int. node) · #internal node ≤ $O(1) \cdot d = O(d)$

Oss Not comparable with the LB on comparison based approach

Prop Using compacted trie #internal node = #branching node = $O(n)$

Radix sort (LSD-first)

martedì 7 marzo 2023 17:29

$L \sim \frac{N}{n}$ avg length $\quad \forall i\ s_i \in S,\ s_i := s_i[1, L]$ (otherwise padding)

```
LSD-first(S) // S = S[1,n][1,L], S[i,:] = s_i
  for i = L, ..., 1 {
      A ← CONCATANATE(s_1[i], ..., s_n[i]); // i-th significant chars (by row)
      (index, A) ← COUNTING SORT(A);
      S ← S[index, :];
  }
```

- Space: $O(N)$
- Time: $O(N + L\sigma)$

(*)

$\in$) cache misses

```
COUNTING SORT(A)
  min ← MIN(A); max ← MAX(A);
  C ← ZEROS(1, max - min); // C = 0, ..., 0
  for i = 1, ..., n
      C[A[i] - min + 1] ++; // # occurrences
  for i = 1, ..., max - min
      OUTPUT(min + (i-1), C[i]); // C[i] times
```

- Time: $O(n + \underbrace{max(A) - min(A)}_{\text{OUTPUT}})$

$min, max, \#occ$

(*)

Lemma $\Sigma = \{0, 1, \dots, \sigma - 1\} \Rightarrow$ Correct, Space is $O(N)$ and time $O(N + L\sigma)$

Proof $Space = Space(S) = O(N)$

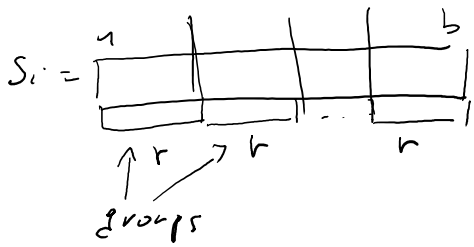
$Time = L \times time(\text{counting sort}) = L \times (n + \sigma) = Ln + L\sigma = N + L\sigma$

Correctness (we prove the stability of Counting Sort): $\alpha, \beta \in S'$ $\alpha < \beta$ and $\alpha = \gamma x \alpha_1$, $\beta = \gamma y \beta_1$ with $\gamma = LCP(\alpha, \beta)$, $x < y$ and $|\alpha_1| = |\beta_1|$

So from right to left (LSD-first) when I compare x and y (after $|\alpha_1| = |\beta_1|$ step) we have $\alpha < \beta$. For the next steps α and β hold the same positions ∎

Optimization: $\forall S_i = B[1, b]$ binary, $\quad 1 \quad\quad b$

Optimization: $\forall S_i = B[1,b]$ binary



Thm Sort n strings of b bits each (using CS to sort r bits at time) with LSD-first $\Rightarrow$ time $= \Theta\left(\frac{b}{r}(n+2^r)\right)$, space $= O(nb) = O(N \lg \sigma)$

Proof Each string S_i of $g = \frac{b}{r}$ "groups" of r bits each.

$\Rightarrow$ time = #digits * time(Counting Sort) = #groups * time(CS)

$= O\left(\frac{b}{r}(n+2^r)\right)$

2^r: max int with r bits

Space $= O(bn) = O(N \lg \sigma)$ bits

#digits, #bit per digit

□

Thm Best choice of r is $r^* = \Theta(\lg n)$ bits, so that time $= O\left(\frac{bn}{\lg n}\right)$ and space $= O(bn)$ bits. Extra space is $\Theta(n)$.

Proof Counting sort time $= \Theta(n+2^r)$ so it is useless choose $r < \lg n$

for $r \in [\lg n, n]$ $\frac{b}{r}(n+2^r) \uparrow \Rightarrow r^* = \lg n$. Time and space substituting r^*

Extra space for Counting Sort (not in place) $\Rightarrow \Theta(|C|) = \Theta(2^{r^*}) = \Theta(n)$ □

Cor time $= O\left(\frac{N \lg \sigma}{\lg n}\right)$, space $= O(N \lg \sigma)$ bits

Proof Recall $bn = N \lg \sigma$

#digits, #bits per digit

□

Cor MSD-first better of LSD-first iff $d = O\left(\frac{N}{\lg n}\right)$

Proof

	time
MSD-first	$O(d \lg \sigma)$
LSD-first	$O\left(\frac{N \lg \sigma}{\lg n}\right)$

Multi-key Quicksort

martedì 7 marzo 2023 20:23

Assume $R = \{s_1, \dots, s_n\}$ prefix free ($\forall i \neq j \; s_i \not\prec s_j \wedge$)
(If not consider $R' = \{s_i\$\} = \{s_i\$\}$, with $\$ \prec a \; \forall a \in \Sigma$)

```
MULTIKEYQS(R, i): //Start i=1
    if |R| ≤ 1
        return R;
    else {
        p ← CHOOSE PIVOT(R);
        (R<, R=, R>) ← THREE-WAY PARTITIONING(R, p, i);
        A ← MULTIKEYQS(R<, i);
        B ← MULTIKEYQS(R=, i+1);
        C ← MULTIKEYQS(R>, i);
        return CONCATANATE(A, B, C);
    }
```

- Extra space: NO
- Time: $O(d + n \lg n)$ on avg (*)

(*) **Thm** It's correct, perform $O(d + n \lg n)$ comparison in avg. If good pivot selection strategy $\Rightarrow O(d + n \lg n)$ time in worst case.

Proof • Correctness: invariant of MULTIKEYQS(R, i): "all strings are sorted up to $(i-1)$-th character".

$(R, i=1) \rightarrow (R_<, i=1)$, $(R_=, i=2)$, $(R_>, i=1)$, ...

• Time on avg.: Let $s \in R$:

$s \in R$ COMPARISON$(s[i], p[i])$ $O(1)$

① $\rightarrow s \in R_< \cup R_> \Rightarrow$ MQS$(R_< \cup R_>, i)$

② $\rightarrow s \in R_= \Rightarrow$ MQS$(R_=, i+1)$

if good pivot selection (balanced partition) $|R_< \cup R_>| \underset{(*)}{\leq} \alpha n$, $\alpha < 1$:

① can occur at most $\log_{\frac{1}{\alpha}} |R| = O(\lg n)$ times

② can occur at most $|s|$ times (d_s exploiting prefix driving)

$\Rightarrow O(d_s + \lg n)$ # comparison for each s

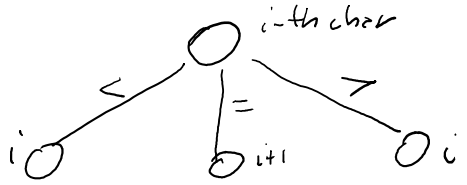
$\Rightarrow \sum_{s \in R} O(d_s + \lg n) = O(d + n \lg n)$ comparisons.

prefix */

$\Rightarrow \sum_{s \in R} O(d_s + \lg n) = O(d + n \lg n)$ comparisons.

Good selection strategy: select the median pivot between $2t+1$ randomly selected pivots (via oversampling).

Multiway-RS or Ternary Search Tree (TST)

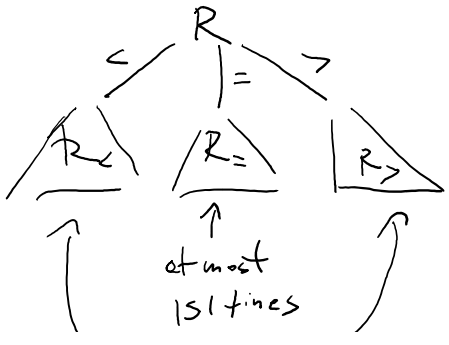



```
// Start: SEARCH(s,T,1)
SEARCH(s,T,i) // T is ternary-search tree (pointer to root root)
  if(s = STRING(T))
      return true;
  if(T=∅ OR i>|s|)
      return false;
  if(s[i] = CHAR(T))
      SEARCH(s, MIDCHILD(T), i+1);
  else if(s[i] < CHAR(T))
      SEARCH(s, LEFTCHILD(T), i);
  else
      SEARCH(s, RIGHTCHILD(T), i)
```

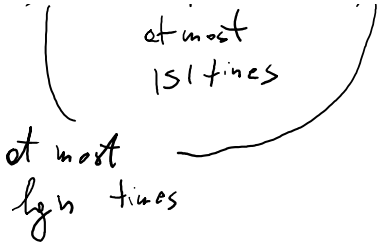
• time: $O(|s| + \lg n)$ (*)

(*) Thm If ternary search tree is perfectly balanced time of search is $O(|s| + \lg n)$

Proof $|R_< \cup R_>| \leq \alpha n$, $\alpha < 1$ $n = |R|$



at most $O(|s| + \lg n)$


at most
|S| times
at most
log n times

General

lunedì 13 febbraio 2023 20:05

$\mathcal{O}$ = {objects}, $\mathcal{U}$ = {all keys} $\quad u = |\mathcal{U}|, n = |\mathcal{O}|$

$\mathcal{O} \longrightarrow S \subseteq \mathcal{U}$ bijective

$x \longmapsto key(x) = k$

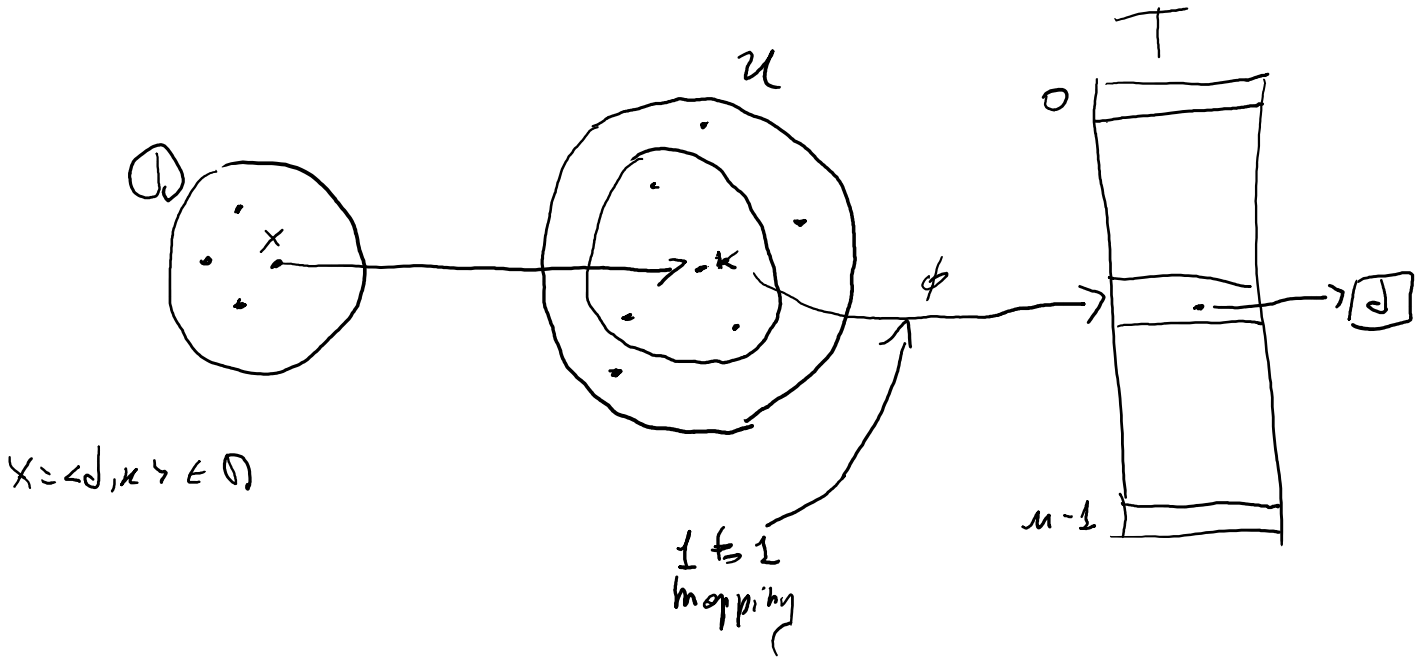
object $x \in \mathcal{O}$, $x = \langle d, k \rangle$

satellite data, $key(x)$

ops: { Search(k), Insert(k), Delete(k)

Direct address table

venerdì 10 marzo 2023 17:15



	Time	Space
Search(k)	O(1)	
Insert(k)	O(1)	O(n)
Delete(k)	O(1)	

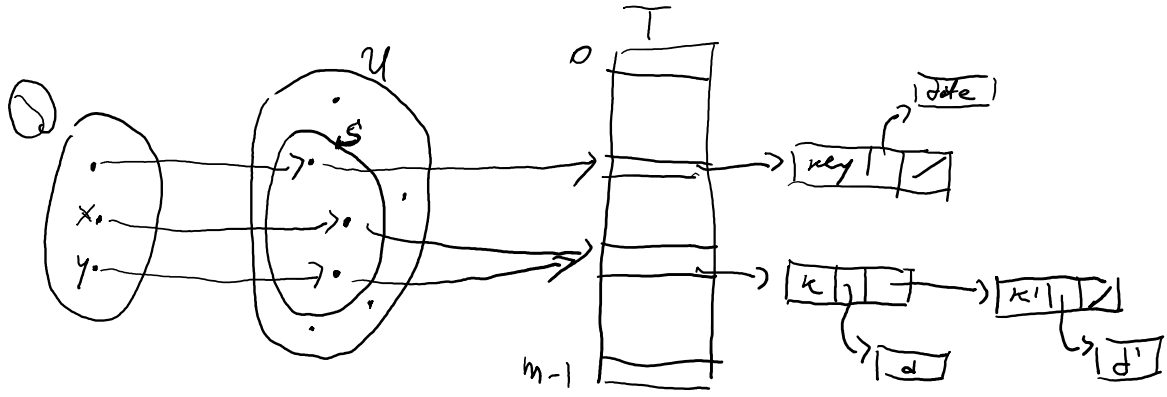
(+) Fast

(−) Space if $u \gg n$

Hash Table (hashing with chaining)

venerdì 10 marzo 2023 17:24

hash function $h : U \longrightarrow \{0, 1, \dots, m-1\}$ $\quad |U| = u \gg m$



Def Collision If $k' \neq k \in U$ and $h(k) = h(k')$

Def Simple uniform hashing $h(k) := k \pmod m = [k]_m$

	Time	Space
Search(k)	on avg	$O((m+n)\lg_2 n + n \lg_2 u)$ b.t.s (*)
Insert(k)	$\Theta(1+n/m)$ (*)	
Delete(k)	α "load factor"	

(*) **Thm** Using $h(k) = [k]_m$ $\exists$ hashtable of size $= m$ such that time of Search(k) is $\Theta(1+\frac{n}{m})$ on avg.

Proof 1st case) $k \notin S$:

$$\text{time on avg.} = \text{time}(\text{compute } h(k)) + E[\text{lenght}(\text{list } T[h(k)])]$$

$\text{time}(\text{compute } h(k)) = O(1)$

$$E[\text{lenght}(T[h(k)])] = \sum_{x \in O} (h(\text{key}(x)) = h(k)) = \sum_{x \in O} \frac{1}{m} = |O| \times \frac{1}{m} = \frac{n}{m} = \alpha$$

2nd case) $x \in S$ $(x = \langle d, k \rangle \in D)$

Let $\ell(x) = T[h(key(x))]$, $E[\text{lenght } \ell(x)] = n_i \simeq \frac{i-1}{m}$, if x is the i-th element inserted in T

$\Rightarrow$ time search $= \frac{1}{n} \sum_{i=1}^{n} (\#\text{items examined in } \ell(x) \text{ with } x \text{ inserted as } i\text{th element}) = \frac{1}{n} \sum_{i=1}^{n} (1 + n_i) = \frac{1}{n} \sum_{i=1}^{n} (1 + \frac{i-1}{m})$

(1 = x itself)

$= 1 + \frac{1}{n} \underbrace{\sum_{i=1}^{n-1} \frac{i}{m}}_{\frac{n(n-1)}{m}} \simeq O(1 + \frac{n}{m})$

(*) Thm Hash table with chaining occupies $(m+n)\lg_2 n + n \lg_2 u$ bits

Proof Space = space(pointers) + space(item keys) (in bits)

$= (\#\text{pointers of } T + \#\text{pointers of chains}) \cdot size(\text{pointer}) + (\#\text{item keys}) \cdot size(\text{keys})$

($\#$pointers of T = m; $\#$pointers of chains: $O(n) = \sum_{i=1}^{n} \frac{n}{m}$; $size(\text{pointer}) = \lg_2 n$ (index one out of n items); $\#$item keys = n; $size(\text{keys}) = \lg_2 u$ (key $\in U$))

$\simeq (m+n)\lg_2 n + n \lg_2 u$

Dynamical hash table:

- Start: $|Q_0| = n_0 \Rightarrow m \leftarrow 2n_0$

Rebuilding technique:
- if $n < \frac{n_0}{2}$: $m \leftarrow \frac{m}{2}$; REBUILD(T, h, m); $n_0 \leftarrow n$
- if $n > 2n_0$: $m \leftarrow 2m$; REBUILD(T, h, m); $n_0 \leftarrow n$

OSS Load factor $\alpha = \frac{m}{n} = 2$

Cor Dynamic hash table with simple uniform hashing takes constant time, expected and amortized for each op.

Proof With maintaining $\alpha = 2$ time for op $= O(1)$.

time(REBUILDING(...)) = time(n item to delete) + time(n item to insert) $= \Theta(n)$

$\Theta(n)$ $\Theta(n)$

at worst case (all insertion or all deletions):

n_2 or $n_1 = \Omega(n)$ times $\Theta(n)$

op(k) + ... + op(k) + REBUILD(...) $\Rightarrow$ Spread $\Theta(n)$ among n operations having cost $= O(1)$

$O(1)$ $\Rightarrow$ $O(1)$ amortized time per op.

□

Universal hashing

venerdì 10 marzo 2023 19:20

Problem: to generate a good hash function, it should be one of all possible hash functions (uniform and independent distr. among keys)

$h: U \to \{0,\dots,m-1\} \Rightarrow m^u$ possible functions $\Rightarrow O(\lg_2 m^u) = O(u \lg_2 m)$ bits

Space to represent it $(-)$

Solution: choose h random in a good set $\mathcal{H}$

Def $\mathcal{H}$ is universal iff:

$$\forall x,y \in U \quad |\{h \in \mathcal{H} \mid h(x) = h(y)\}| \le \frac{|\mathcal{H}|}{m}$$

Def c-universal if-f $\le c\frac{|\mathcal{H}|}{m}$

Prop $h \in \mathcal{H}$ universal, $\forall x,y \in U \quad p(h(x) = h(y)) \le \frac{|\mathcal{H}|/m}{|\mathcal{H}|} = \frac{1}{m}$ $\le \frac{c}{m}$ if c-universal

Thm Hash table $T[0,m-1]$, with $h \in \mathcal{H}$ universal. For a generic (key) $k \in S$ $E[\text{lenght}(l_{h(k)})] \le 1+\alpha$, with $\alpha = \frac{n}{m}$

Proof $\forall k,k' \in S$

$$I_{k,k'} := \begin{cases} 0 & \text{if } h(k) \neq h(k') \\ 1 & \text{otherwise} \end{cases} \qquad \left(p(I_{k,k'} = 1) \le \frac{1}{m}\right) (*)$$

Let $N_k = |\{k' \in S \mid k' \neq k, h(k') = h(k)\}|$, $N_k = \sum_{\substack{k' \neq k \\ k' \in S}} I_{k,k'}$

$$E[\text{lenght}(l_{h(k)})] = E[N_k + 1] = E[N_k] + 1 = E\left[\sum_{k' \neq k} I_{k,k'}\right] + 1 =$$

($+1$: we must consider k)

$$= \left(\sum_{k' \neq k} E[I_{k,k'}]\right) + 1 =$$

$$= \left(\sum_{k' \neq k} p(I_{k,k'} = 1)\right) + 1 \le \frac{n-1}{m} + 1 \le 1+\alpha$$

($I_{k,k'}$ I.R.V.; $(*)$)

Thm $T[0,m-1]$ hash table with $h \in \mathcal{H}$ universal. If we insert $n = \Theta(m)$ keys
$\Rightarrow E[\text{length of longest chain}] = O\left(\frac{\lg n}{\lg \lg n}\right)$

(NO PROOF)

THE POWER OF TWO CHOICES

d sub-tables $T_1, \dots, T_d$ of size m/d with respectively d hash functions
$h_1, \dots, h_d : U \to \{0, \dots, \frac{m}{d} - 1\}$

```
INSERT(x, T_1,...,T_d, h_1,...,h_d):
    for i = 1,...,d:
        a_i ← LENGHT(T_i[h_i(x)]);
    j ← ARGMIN(a_1,...,a_d); // leftmost in case of tie
    INSERT(x, T_j, h_j);
```

• time: $O(d)$

```
SEARCH(x, T_1,...,T_d, h_1,...,h_d)
    i ← 1; stop ← false;
    while (!stop AND i < d) {
        if (SEARCH(x, T_i, h_i))
            stop = true;
        else
            i++;
    }
    return stop;
```

• time $= \begin{cases} O(\lg \lg n) & \text{if } d = 2 \\ O\left(d\left(\frac{\lg \lg n}{\lg d} + 1\right)\right) & \text{if } d > 2 \end{cases}$

$d \cdot$ time(SEARCH(T_i)) $\geq d \cdot E[\text{length of chain}]$
$= d \cdot \left(\frac{\lg(\lg n)}{\lg d} + O(1)\right)$

Examples of universal $\mathcal{H}$

1) Assume $m = 101$ prime.

$x \in U$, $x = [\underbrace{x_0}_{\lg_2 m \text{ bits}}, \dots, x_{r-1}]$ $\lg_2 u$ bits. $r = \frac{\lg_2 u}{\lg_2 m}$ $\left(|U| = m^r\right)$

$a \in \{1, \dots, m-1\}$, $a = [a_0, \dots, a_{r-1}]$ (as x)

$$\mathcal{H} = \left\{ h_a(k) = \sum_{i=0}^{r-1} a_i k_i \pmod m \right\}_{a \in \{1,\dots,u-1\}}$$

Thm $\mathcal{H}$ is an universal class

Proof Let $x \neq y \in U$. Without loss of generality we suppose $x_0 \neq y_0$

$$h_a(x) = h_a(y) \iff \sum_{i=0}^{r-1} a_i x_i \equiv \sum_{i=0}^{r-1} a_i y_i \pmod m$$

$$\iff a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r-1} a_i (x_i - y_i) \pmod m$$

$$\iff a_0 \equiv \left[-\sum_{i=1}^{r-1} a_i (x_i - y_i) \right] (x_0 - y_0) \pmod m$$

($\iff$: $x_0 \neq y_0$ and m prime)

$\Rightarrow$ 1 choice for a_0 and $a_1, \dots, a_{r-1}$ free

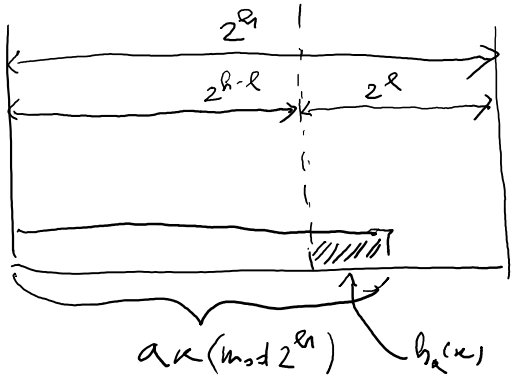
$\Rightarrow$ $|\{ h_a(x) = h_a(y) \mid h_a \in \mathcal{H} \}| = m^{r-1} = \frac{u}{m} = \frac{|U|}{m}$ ▨

2) $m < |U|$ size of T (m not prime). Set $p > |U|$ prime

$$\mathcal{H}_{pim} = \left\{ h_{a,b}(k) = ((ak+b) \bmod p) \bmod m \right\}_{\substack{a>0 \\ b \geq 0}}$$

3) Assume $|U| = 2^h$, $m = 2^l < |U|$, $a \equiv 1 \pmod 2$ and $a < |U|$

$$\mathcal{H}_{h,l} = \{ h_a(k) = (ak \bmod 2^h) \operatorname{div} 2^{h-l} \}$$



Thm $\mathcal{H}_{h,l}$ is 2-universal ($\forall x \neq y \in U$, $h_a \in \mathcal{H}_{h,l}$ $p(h_a(x) = h_a(y)) \leq \frac{2}{m}$)

Thm $\mathcal{H}_{bit}$ is 2-universal $\left(\forall x \neq y \in U,\ h_a \in \mathcal{H}_{bit}\quad p(h_a(x) = h_a(y)) \leq \frac{2}{m}\right)$

(NO PROOF)

Perfect hashing, minimal, ordered

lunedì 13 marzo 2023 20:30

Def $h: U \to \{0,\dots,m-1\}$ (PHF) perfect w.r.t. a dictionary $S \subseteq U$ iff $\forall x \neq x' \in S$ $h(x) \neq h(x')$

oss h perfect $\Rightarrow |D| = |S| = n \leq m$

Def If $m = n$, h perfect $\iff$ h minimal (MPHF) perfect

oss h MPHF $\Rightarrow$ Search(x), Insert(x) (with hash table), Delete(x)

- Space: $O(n)$
- Time: $O(1)$

Def h is order preserving (OPHF) w.r.t. S iff $\forall x < x'$ in $S \Rightarrow h(x) < h(x')$

oss h order preserving $\Rightarrow h$ perfect

Def If $|S| = n = m$ and h OPHF $\iff h$ OPMPHF

Construction of a OPMPHF (Supposing fixed S)

- Pick randomly h_1, h_2 from $\mathcal{H}$ universal,

$$h_1, h_2 : U \longrightarrow \{0,\dots,m'-1\} \quad \text{with } m' = cn \text{ (tipically)}, \ c > 1$$

$(m' > |S| = |D|)$

- Design a function $g : \{0,\dots,m'-1\} \longrightarrow \{0,\dots,n-1\}$ $(m' \geq n)$

s.t. $h(t) := g(h_1(t)) + g(h_2(t)) \pmod n$ is OPMPHF ($h: U \to \{0,\dots,n-1\}$ OPMPHF w.r.t. S)

Start: Graph $G = (V, E)$, $V = \{0,\dots,m'-1\}$, $E = \{(h_1(t), h_2(t))\}_{t \in \{0,\dots,n-1\}}$ $(|V| = m', \|E\| = n)$

Ass adjacency list of $i \in \{0,\dots,m'-1\}$

$\forall (u,v) \in E$ $(u, v \in \{0,\dots,m'-1\})$ has weight $h(u,v) = t$ where $u = h_1(t)$ and $v = h_2(t)$

"on edge is excess"

```
LABEL_ACYCLY_GRAPH(G)
    for v ∈ V
        g[v] = undef // initialize labels of nodes
```

- Extra space: $\Theta(n)$
- Time: $\Theta(n)$ on ave. (*)

```
for v∈V
    g[v] = undef // initialize labels of nodes
for v∈V {
    if g[v] = undef
        LABEL_FROM(v, 0);
}

LABEL_FROM(v, c)
  if g[v] ≠ undef {
      if g[v] ≠ c
          return; // the graph is cyclic
  }
  g[v] = c; // new label of node v
      for u ∈ A[v]
          LABEL_FROM(u, h(u,v) - g[v] mod n);
```

- extra space: $\mathcal{H}(n)$
- Time: $\mathcal{H}(n)$ on avg. (*)

Note: If it returns a graph with unlabeled nodes (cyclic graph case) pick two new hash functions, rebuild G_1 and restart the alg.

Prop (*) To construct the OPMPHF we need to build $\mathcal{H}(1)$ graphs on avg. So extra space and time are $\mathcal{H}(n)$ in avg.

Proof Given $|V| = m'$ and $|E| = n$

$$P(\text{build acyclic graph}) \approx \begin{cases} 0 & \text{if } m' \le 2n \text{ (few nodes)} \\ \underbrace{\frac{m'-2n}{m'}}_{p} & \text{if } m' > 2n \end{cases}$$

$$\Rightarrow E[\#\text{graph built}] = \sum_{l=1}^{\infty} l \cdot (1-p)^{l-1} \cdot p = \frac{1}{p} \qquad \left(P(\#\text{graph built} = k) = (1-p)^{k-1} p\right)$$

↑ geometric distribution

$$= \sqrt{\frac{m'}{m'-2n}} \overset{\uparrow}{=} \mathcal{H}(1)$$

$m = \mathcal{H}(n)$

$\Rightarrow$ Time on avg. $= \mathcal{H}(1) \cdot \underbrace{\text{time}(\text{alg})}_{n} = \mathcal{H}(n)$

$\Rightarrow$ Space on avg. $= \mathcal{H}(1) \cdot \text{space}(\text{one graph}) = \mathcal{H}(1) \cdot (\underbrace{m'}_{\text{nodes}} + \underbrace{n}_{\text{edges}}) \overset{\uparrow}{=} \mathcal{H}(n)$

$m' = \mathcal{H}(n)$

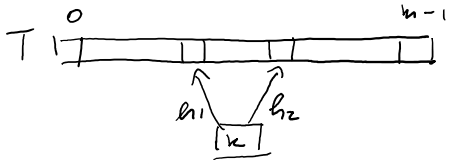
□

Cuckoo hashing

martedì 14 marzo 2023 01:38

(efficient for dynamic dictionary)

$h_1, h_2 : U \longrightarrow \{0, \dots, m-1\}$




```
SEARCH(k, T, h1, h2):
    i1 ← h1(k); i2 ← h2(k);
    if(! SEARCH(k, T[i1]))
        return SEARCH(k, T[i2])
    else
        return true;
```

```
DELETE(k, T, h1, h2):
    i1 ← h1(k); i2 ← h2(k);
    if(SEARCH(k, T[i1]))
        return DELETE(k, T[i1]);
    else if(SEARCH(k, T[i2]))
        return DELETE(k, T[i2]);
    else
        return T;
```

	time	space
Search(k)	$O(1)$	(*) $\Omega(2^{th} \cdot 6^{keys})$ bits
Delete(k)		
Insert(k)	$O(1)$ amortized expected (*)	

th: $\log_2 m$ bits; keys: $\log_2 u$ bits

```
INSERT(k, T, h1, h2, max)
    if(T[h1(k)] = ϕ){
        T[h1(k)] = k;
        return T;
    }
    if(T[h2(k)] = ϕ){
        T[h2(k)] = k;
        return T;
    }
    i ← T[h1(k)]; //w.l.o.g. we could choose T[h2(k)]
    steps ← 1;
    while(T[i] ≠ ϕ AND steps < max)
        {T, k, i ← KICKOUT(T[i], k); steps++}
    if(steps ≥ max)
        return false; //we must replace two new h1, h2
    else
        {T[i] ← k; return T;}
```

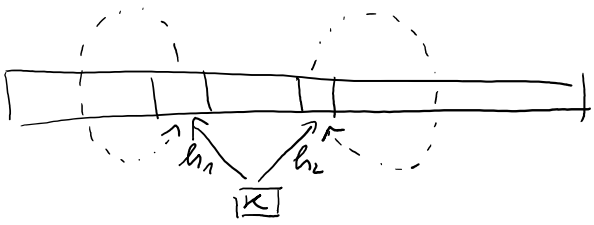
```
KICKOUT(T[i], k);
    k' ← T[i];
    j ← hx(k'); //alternative pos. of k' (i → j ∈ {1,2})
    T[i] ← k;
    return T, k', j;
```

$T[i,j] \leftarrow k$;
return T, k', j';

Def Cuckoo graph $G=(V,E)$ direct graph

$V=\{i \mid T[i] \neq \phi\}$, $E=\{i \to j \mid i=h_x(k), j=h_y(k) \text{ and } T[i,j]=k\}$ $(x \neq y \in \{1,2\})$

oss Infinite loop iff



Def $k' \neq k$, $k' \in bucket(k)$ iff $\exists$ undirected path from $h_1(k)$ or $h_2(k)$ to $h_1(k')$ or $h_2(k')$ $\left(\exists\ h_i(k)/h_2(k) \overset{\text{u.path}}{\sim} h_1(k')/h_2(k')\right)$

Thm $\forall i,j, \forall c>1$ if $m \geq 2cn \Rightarrow p(i \overset{L}{\sim} j, \text{shortest}) \leq \frac{1}{c^L m}$ $(L \geq 1)$

Proof By induction:

$L=1$) $p(i \overset{1}{\sim} j) = p(i \to j \vee j \to i) \leq \frac{2}{m^2}$ $\left(p(i \to j) = \frac{1}{m^2}\right)$ (union bound)

$\Rightarrow \sum_{k \in S}^{i} \frac{2}{m^2} = \frac{2n}{m^2} \leq \frac{1}{cm}$ $\left(n \leq \frac{m}{2c}\right)$

$L-1 \Rightarrow L$) $p(i \overset{L}{\sim} j) = P(\exists z \mid i \overset{L-1}{\sim} z \overset{1}{-} j) \leq$ (union bound) $m \cdot p(i \overset{L-1}{\sim} z \overset{1}{-} j) =$ (m: choices of z)

$= m \cdot p(i \overset{L-1}{\sim} z) \cdot p(z \overset{1}{-} j) \leq m \cdot \frac{1}{c^{L-1} m} \cdot \frac{1}{cm} = \frac{1}{c^L m}$ (inductive step)

∎

Thm $p(k' \in bucket(k)) = O(\frac{1}{m})$, given $k, k' \in S$ (with $m \geq 2cn$)

Proof $p(k' \in bucket(k)) = p(\exists L \mid h_i(k) \overset{L}{\sim} h_j(k')_{i,j \in \{1,2\}}) \leq 4 \cdot \sum_{L=1}^{\infty} \frac{1}{c^L m} = \frac{4}{m} \cdot \frac{1}{c-1} = O(\frac{1}{m})$

(4 possible choice of path)

Thm $p(\text{successful insertion}) \leq \frac{1}{c-1}$ (with $m \geq 2cn$)

Proof $p(\text{infinite loop}) = p(\exists i, j : \text{(i)}, \text{(j)}) \leq p(\exists i : \text{(i)})$

$= p(\exists i, \exists L : \text{(i)}L) = \sum_{i=0}^{m-1} \sum_{L \geq 1} \frac{1}{m c^L} = m\left(\frac{1}{m} \cdot \sum_{L \geq 1} \frac{1}{c^L}\right)$

$\longrightarrow \frac{1}{c-1}$ ∎

Corollary 1 Setting $c=3$ $\underbrace{p(\text{infinite loop})}_{1-p} \leq \frac{1}{2}$ $(\text{with } m \geq 2cn)$

Corollary 2 $E[\#\text{rehashes}] = \sum_{i} i\,(1-p)^{i-1} p = \frac{1}{p} \leq 2$ $(p \geq \frac{1}{2}$ prob of succesfull insertion$)$

Corollary 3 Suppose to perform εn insertion in a table hash of size m with initially n keys and suppose $m \geq 2c \cdot (n + n\varepsilon)$. The cost of the εn insertion has constant expected amortized time.

Proof Using Cor 1 and Cor 2 $(\text{with } m \geq 2c(n+n\varepsilon))$ we have costant number of re-hashes expected

$E[\text{time}] = E[\#\text{re-hashes}] \cdot \text{time(rehash)}$

$= \Theta(1) \cdot \underbrace{O(n)}_{\text{compute 2 hashes per key}} = O(n)$ (for all $n\varepsilon$ insertions)

$\Rightarrow O\left(\frac{n}{n\varepsilon}\right) = O\left(\frac{1}{\varepsilon}\right) = O(1)$ expected amortized time for one insertion ∎

OSS For general case (generic n and ε) use rebuilding technique (half or double the table)

Bloom filters

martedì 14 marzo 2023 16:59

When storage of the keys is expensive (if $n \log_2 u$ bits is too large)

(+) keys not explicitly stored

(-) one-side error (only FP errors)

$h_i : U \longrightarrow \{0, 1, \dots, m-1\}$ $\quad i = 1, \dots, r$ universal hash functions

```
// Start: B = ZEROS(1, m)
INSERT(k, B, h_1, ..., h_r)
    for i = 1, ..., r
        B[h_i(k)] ← 1
```

```
SEARCH(k, B, h_1, ..., h_r) // possible FP error
    res ← true; i ← 1;
    while(res AND i < r){
        if(B[h_i(k)] = 0)
            res ← false;
    }
    return res;
```

	Time	Space
Insert(k)	$O(r)$	m bits
Search(k)	$\overset{(*)}{=} \frac{m}{n} \ln 2$	
Delete(k)		

Lemma $p(B[j] = 1) = 1 - e^{-\frac{nr}{m}}$

Proof $p(B[j] = 0) \overset{n \text{ keys}, r \text{ hashes}}{=} \left(\frac{m-1}{m}\right)^{nr} = \left(1 - \frac{1}{m}\right)^{nr} = \left[\left(1 - \frac{1}{m}\right)^{m}\right]^{\frac{nr}{m}} \overset{\lim_{n \to +\infty} \left(1 + \frac{x}{n}\right)^n = e^x}{\approx} \left[e^{-1}\right]^{\frac{nr}{m}} = e^{-\frac{nr}{m}}$

$\Rightarrow p(B[j] = 1) = 1 - p(B[j] = 0) = 1 - e^{-\frac{nr}{m}}$ ▨

Prop $p(\text{FP error}) = \left(1 - e^{-\frac{nr}{m}}\right)^r \doteq p_{err}$

IVoS $p(\text{FP error}) = (1 - e^{-\frac{rn}{m}})^r \simeq p_{err}$

Proof $p(\text{FP error}) = p(x \notin S, B[h_1(x)] = \dots = B[h_r(x)] = 1)$

$= p(B[j_1] = \dots = B[j_r] = 1)$

$= (p(B[\hat{j}] = 1))^r = (1 - e^{-\frac{rn}{m}})^r$ ☒

Corollary (*) given n and m fixed the best number of hashes is $r^* = \frac{m}{n} \ln 2$ and the respective p_{err} optimal is $(0.6185)^{m/n}$.

Proof $f(r) := (1 - e^{-\frac{rn}{m}})^r$ $\quad f'(r) = 0 \iff r = \frac{m}{n} \ln 2$

$\Rightarrow p_{err} = (1 - e^{-\frac{nr^*}{m}})^{r^*} = (1 - e^{-\ln 2})^{\frac{m}{n} \ln 2} = \left[\left(\frac{1}{2}\right)^{\ln 2}\right]^{m/n} = (0.6185)^{m/n}$ ☒

Prop To ensure $p_{err} \leq \varepsilon$ with ε given we must have B of size $\hat{m} \geq 1{,}44 \cdot \lg_2\left(\frac{1}{\varepsilon}\right) n$ bits

Proof We want $p_{err} \leq \varepsilon \iff \left[\left(\frac{1}{2}\right)^{\ln 2}\right]^{m/n} \leq \varepsilon$ (using r^*)

$2^{(\ln 2 \cdot \frac{m}{n})} \geq \frac{1}{\varepsilon}$

$\ln 2 \cdot \frac{m}{n} \geq \lg_2\left(\frac{1}{\varepsilon}\right)$

$m \geq \frac{n}{\ln 2} \lg_2\left(\frac{1}{\varepsilon}\right) \approx 1{,}44 \cdot \lg_2\left(\frac{1}{\varepsilon}\right) \cdot n$ ☒

Spectral Bloom filters

martedì 14 marzo 2023 18:38

Extention of bloom filters to multi-sets

Def $S = \{(x, f_x)\}$ multiset $x \in U$, f_x = multiplicity of x in S

h_{is} $h_1, \dots, h_z : U \longrightarrow \{0, \dots, m-1\}$

Counting counter vector $C[i] = \sum_{x \text{ s.t. } h_j(x)=i,\ j=1,\dots,r} f_x$

Ops: { Counting(x) / Insert(x)/Increment(x) / Delete(x)/Decrement(x)

```
//start: C = zeros(1,m)
INCREMENT(x, f_x, C, h_1, ..., h_z)
    for i = 1, ..., z
        C[h_i(x)] += f_x;

DECREMENT(x, f_x, C, h_1, ..., h_z)
    for i = 1, ..., z
        C[h_i(x)] -= f_x;

COUNTING(x, C, h_1, ... h_z) // overestimates f_x in S
    min ← 0;
    for i = 1, ... r // C[h_i(x)] are overestimations
        if (C[h_i(x)] < min)
            min ← C[h_i(x)];
    return min;
```

	time	space
Increment(x)	$z = \frac{m}{n} \ln 2$	32 · m bits (32: to store integer in C[·])
Decrement(x)		
Counting(x)		

Prop $p(\text{error counting}) = (0.6185)^{m/n}$

Proof $p(\text{error}) = p(\text{larger estimation}) = p(c[h_i(u)] > f_u \ \forall i = 1, \dots, z)$

$= p(\text{collision } \forall i = 1, \dots, z) \underset{\text{see BF using } r^*}{\uparrow} \simeq \left(1 - e^{-\frac{nr}{m}}\right)^{z^*} = \left(\frac{1}{2}\right)^{z^*} = \left[\left(\frac{1}{2}\right)^{\rho_{hz}}\right]^{\frac{m}{n}} \simeq (0.6185)^{\frac{m}{n}}$

General

lunedì 13 febbraio 2023 20:06

$D = \{s_1, \dots, s_n\}$ dictionary of strings, Σ alphabet

$n = |D|$

$N = \sum_{s_i \in D} len(s_i)$

$|\Sigma| = \sigma$

We use 2 special characters \$ and # such that

$\$ < a \quad \forall a \in \Sigma$

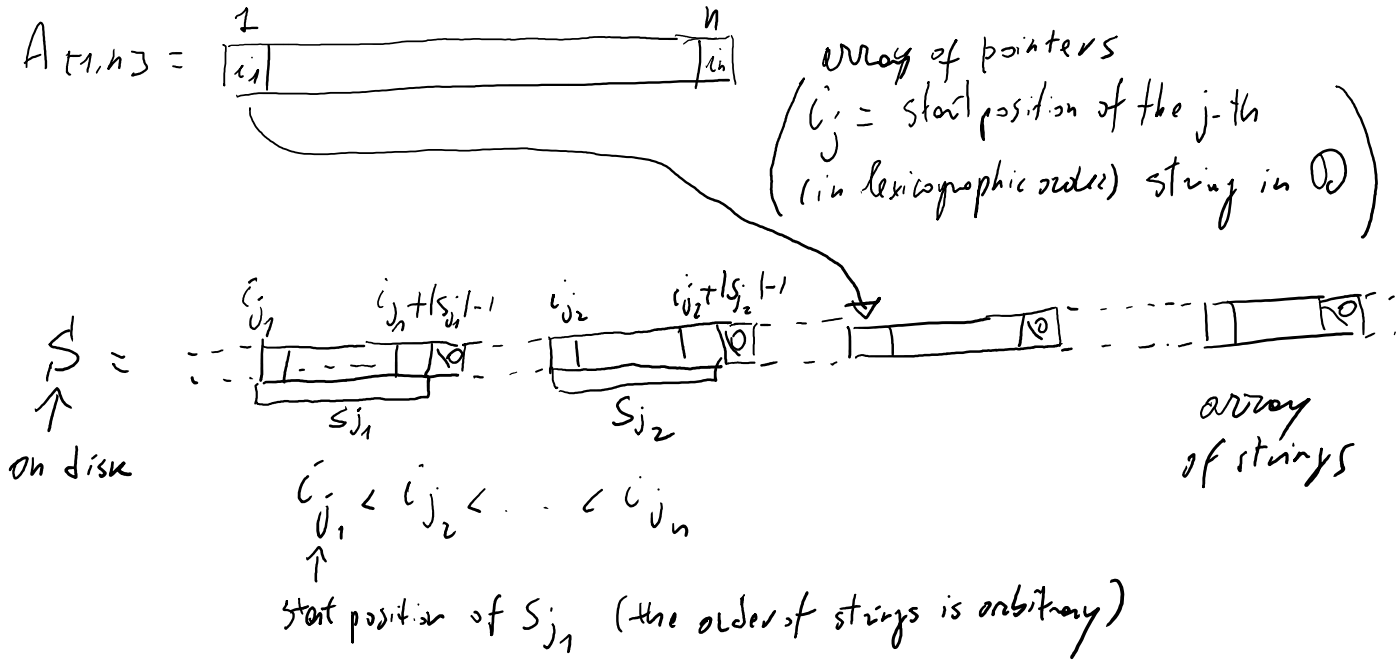
$\# > a \quad \forall a \in \Sigma$

2 Operations given string $P[1,p]$

- $Count(P) = |\{s_i \in D \mid s_i = [P, q]\}|$
- $Retrieve(P) = \{s_i \in D \mid s_i = [P, q]\}$

Array of string pointers

martedì 14 marzo 2023 19:39




```
RETRIEVE(P, A, S):
    l ← BINARYSEARCH(P$, A, S);
    r ← BINARYSEARCH(P#, A, S);
    return A[l,r].
```

- Space: $N\lg\sigma + (\lg\sigma + w)n$ bits (32/64 bits)
- Time: $O(p \lg n)$
- I/Os: $O(\frac{p}{B}\lg n)$

(4)

Thm time is $O(p \lg n)$, I/Os are $O(\frac{p}{B}\lg n)$ and the total space is $N + (1+w)n$ bits. Retrieve n_{occ} prefixed by P needs $\Omega(n_{occ})$ I/Os. (←)

Proof time: Binary search on A: $O(p \lg n)$ (lexicographic comparison $P[1,p]$; # steps)

Some reasons for $O(\frac{p}{B}\lg n)$ I/Os.

Space: $N\lg\sigma + n\lg_2 + wn$ bits (strings; delimiters; pointers; for S; for A)

$= N + (1+w)n$ bits

After obtained the range $A[l,r]$ each string (prefixed by P) can be visualized using at least 1 I/O because the contiguity in A doesn't imply the contiguity in S

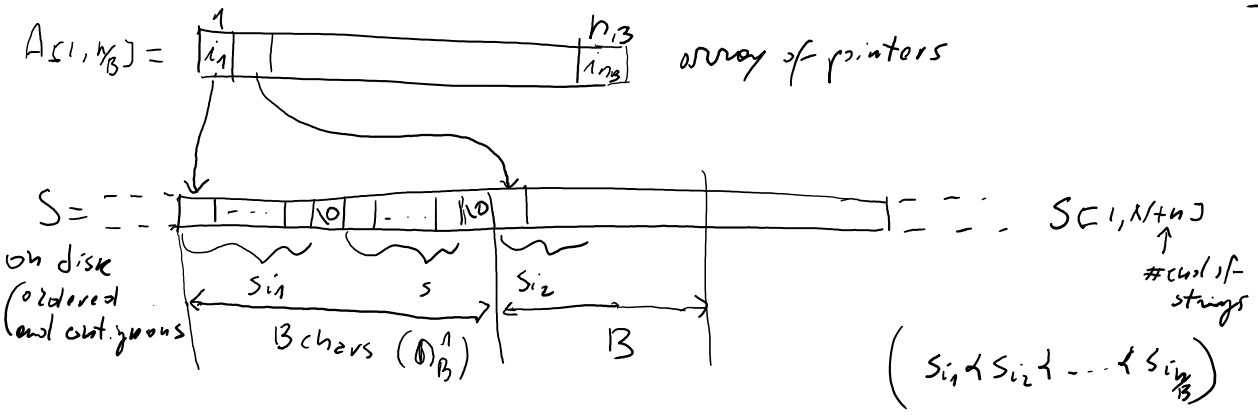
using at least 1 I/O because the contiguity in A doesn't imply the contiguity in S.

Contiguous allocation of strings

martedì 14 marzo 2023 20:53

$\mathcal{D} \rightsquigarrow \mathcal{D}_B = \{s_{i_1}, \ldots, s_{i_{n_B}}\} \subseteq \{s_1, \ldots, s_n\}$ $\quad |\mathcal{D}_B| = n_B \leq \frac{N}{B}$

($s_{i_j} \in \mathcal{D}_B$ iff s_{i_j} is the first string of the j-th block)




```
RETRIEVE(P, A, S) // P[1,p]
    l ← SEARCH(P$, A, S);
    r ← SEARCH(P#, A, S);
    return S[l, r-1];
```

- Space: $(N+n)\lg\sigma + w\, n_B$ bits
- Time: $O(p \lg \frac{N}{B})$
- I/Os: $O(\frac{p}{B} \lg \frac{N}{B})$

(*)

```
SEARCH(Q, A, S); // Q[1,p]
    j ← BINARY SEARCH(Q, A); // give the j-th block
    pos ← SCAN & COMPARE(Q, S[A[j], A[j+1]-1]) // scan & compare Q within the block
    return pos;
```

(*)

Thm Time is $O(p \lg \frac{N}{B})$, I/Os are $O(\frac{p}{B} \lg \frac{N}{B})$ and space is $(N+n)\lg\sigma + w\, n_B$ bits.
Retrieve the strings prefixed by P needs $O(\frac{N_{occ}}{B})$ I/Os (N_{occ} = total length of strings prefixed by P)

Proof time: $O(p \lg n_B + B) = O(p \lg n_B) \leq O(p \lg \frac{N}{B})$

- p: string comparison
- $\lg n_B$: # steps for BS in A
- B: scan the block
- $\leq$: # of word 1 string for each block

for the same reason $O(\frac{p}{B} \lg \frac{N}{B})$ I/Os.

Space: $(N+n)\lg\sigma + w\, n_B$ bits.

Space: $(N+n)\log\sigma + w\frac{n}{B}$ bits.

- $(N+n)\log\sigma$ → S
- w → # bits to store a pointer
- $\frac{n}{B}$ → # pointers in A

After we retrieve $S[l,r]$ we scan the contiguous portion to visualize all the strings with prefix P => $\Omega\left(\frac{N_{occ}}{B}\right)$ ($N_{occ} = r - l$)

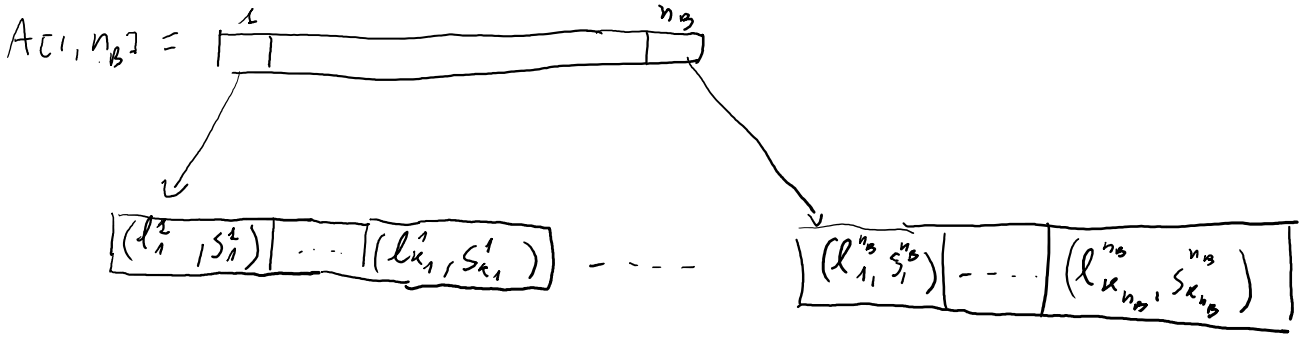
Front coding with bucketing

mercoledì 15 marzo 2023 00:41

$$\mathcal{D} = \{s_1, \ldots, s_n\} \xrightarrow{\Delta\text{-compression}} \{(\ell_1, \hat{s}_1), \ldots, (\ell_n, \hat{s}_n)\}$$

$$\begin{cases} \ell_1 = 0 \\ \ell_i = LCP(s_i, s_{i-1}) \end{cases}, \quad \begin{cases} \hat{s}_1 = s_1 \\ \hat{s}_i = s_i[\ell_i + 1, |s_i|] \end{cases} \Rightarrow s_i = [s_{i-1}[1, \ell_i], \hat{s}_i]$$

Two-level indexing : Front coding per bucket (FC_B)



Def. $FC_B(\mathcal{D})$ = Space to store all the strings compressed by FC_B

Oss Now $n_B \approx \frac{FC_B(\mathcal{D})}{B} < \frac{N}{B}$ (+) (potentially increase the # strings stuffed in a disk page)

RETRIEVE(P, A, S) // P[1, p]
// like contiguous allocation of strings

- Space: $FC_B(\mathcal{D})$ (strings) + $(1+w)\, n_B$ (pointers)
- Time: $O\left(p \lg_2 \frac{FC_B(\mathcal{D})}{B} + B\right)$
- I/Os: $O\left(\frac{p}{B} \lg_2 \frac{FC_B(\mathcal{D})}{B}\right)$

} (*)

(*) Thm I/Os of prefix search are $O\left(\frac{p}{B} \lg_2 \frac{FC_B(\mathcal{D})}{B}\right)$. Retrieving the strings prefixed by P needs $O\left(\frac{FC_B(\mathcal{D}_{occ})}{B}\right)$ I/Os where $\mathcal{D}_{occ}$ are the strings wanted

Proof time $O\left(p \lg_2 n_B + \underbrace{size(decompress\_all(B))}_{SCAN}\right) \approx O\left(p \lg_2 \frac{FC_B(\mathcal{D})}{B} + B\right)$

- p: string comparison
- $\lg_2 n_B$: # steps of BS in A
- $\approx$: oss
- B: usually (can be $O(B^2)$ very rare)

For I/Os: $O(p \lg \ldots)$

For I/Os: $O\left(\frac{P}{B} \lg_2 \frac{FC_B(D)}{B}\right)$.

To retrieve all the strings prefixed by P ($\forall s \in D_{doc}$) we need

$O\left(\frac{FC_B(D_{doc})}{B}\right)$ I/Os $\left(\frac{\text{space to store compressed strings}}{\text{size of disk page}}\right)$

Interpolation search

mercoledì 15 marzo 2023 02:01

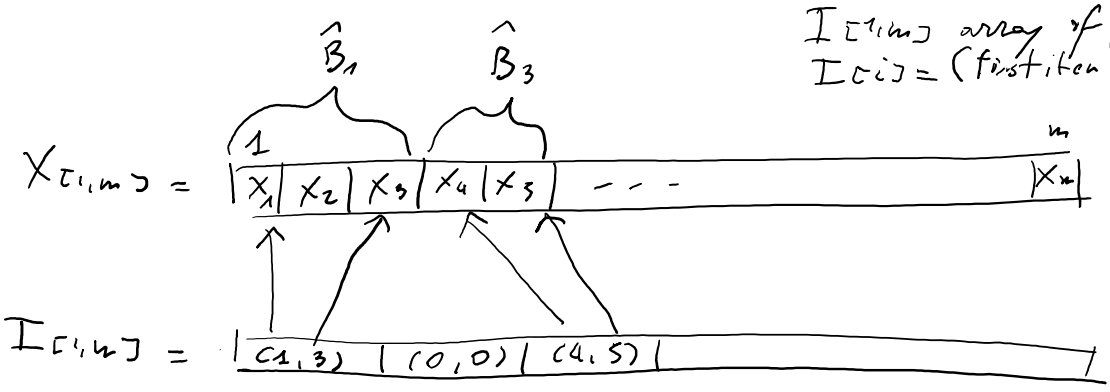
$\mathcal{D}_B$ es integers (if not see string as number in base $\sigma = |\Sigma'|$)

$\mathcal{D}_B \longrightarrow X[1,m] = x_1, \dots, x_m$ (distinct) integers $\quad x_i < x_{i+1} \quad (m \le n_B)$

Transform range $[x_1, x_m]$ in m bins $B_1, \dots, B_m$ of length $b = \frac{x_m - x_1 + 1}{m}$

$$B_i := [x_1 + (i-1)b, x_1 + ib), \quad \hat{B}_i = X \cap B_i$$

$I[1,m]$ array of pointers
$I[i] = (\text{first item of } \hat{B}_i, \text{last item of } \hat{B}_i)$




```
SEARCH(y, X, I): // search integer y in X
    j ← ⌊(y − x_1)/b⌋ + 1; // compute the B_j s.t. y ∈ B_j
    l, r ← I[j]; // determine the pos. of first and last item
    return BINARY SEARCH(y, X[l, r]);
```

- Extra space: $O(m)$
- Time: $O(\lg \Delta)$ (*)

(*) **Thm** Searching a string in a dict of size m, takes $O(\lg \Delta)$ time, where $\Delta = \frac{\max_{i=2,\dots,m}(x_i - x_{i-1})}{\min_{i=2,\dots,m}(x_i - x_{i-1})}$. The extra space is $O(m)$

Proof time: $O(1 + \lg_2 |\hat{B}_j|) = O(\lg_2 |\hat{B}_j|) \quad (< O(\lg_2 b))$

- 1: select B_j
- $\lg_2 |\hat{B}_j|$: BS on $X[l,r]$ with size $|\hat{B}_j| \le |B_j| = b$

A generic $\hat{B}_j$ contains at most $\frac{b}{s}$ integers where $s = \min_{i=2,\dots,m}(x_i - x_{i-1})$.

And

$$\max_{i=2,\dots,m}(x_i - x_{i-1}) \ge \frac{\sum_{i=2}^{m}(x_i - x_{i-1})}{m-1} = \frac{x_m - x_1}{m-1} \ge \frac{x_m - x_1 + 1}{m} = b$$

$\Rightarrow |\hat{B}_j| \le \frac{b}{s} \le \Delta \Rightarrow$ time $= O(\lg_2 |\hat{B}_j|) = O(\lg_2 \Delta)$. $\quad \frac{a}{b} \ge \frac{a+1}{b+1}$ (if $a \ge b$)

$S_{pace} = \underbrace{S_{pace}(X)}_{O(m)} + \underbrace{S_{pace}(I)}_{O(m)} = O(m)$ ∎

Oss $\Delta = \begin{cases} 1 & \text{if unif. distr.} \\ >1 & \text{otherwise} \end{cases}$ and time is $O(\lg(\min\{\Delta, m\}))$ ← worst case

$\Rightarrow \underbrace{O(\lg\lg m)}_{(*)} \le O(\lg \Delta) \le O(\lg m)$ ← time BS on X

(*) Thm If X is uniformly distributed $\Delta = \text{poly}\lg(m)$ with high probability, and time become $O(g(\lg m))$ with high probability.

Proof (Hint) $\Delta = \text{poly}\lg(m) = p(\lg(m)) = a_k \lg^k(m) + \dots + a_1 \lg(m) + a_0$

$\Rightarrow O(\lg_2 \Delta) = O(\lg_2(a_k \lg^k(m) + \dots)) = O(\lg_2(a_k \lg^k(m)))$

$= O(\underbrace{\lg_2 a_k}_{O(1)} + k \lg_2(\lg_2(m))) = O(\lg_2(\lg_2(m)))$ ∎

Compacted trie

mercoledì 15 marzo 2023 15:25

$O = \{s_1, \ldots, s_n\}$, Σ alphabet

Def A trie is a σ-ary search tree ($\sigma = |\Sigma|$). Labels of edges are characters $\in \Sigma$ and labels of internal nodes are integers representing partial length of the substring. Leaves are strings of O. Given a int. node u, $s(u)$ is the string obtained concatenating the the labels of path root $\rightsquigarrow u$.

Def A compacted trie CT is a trie without unary internal nodes.

Oss For a CT: no internal unary node $\Rightarrow$ # internal node $<$ # leaves

Optimization:
- the substring (label of compacted trie edges) can be represented in $O(1)$ space: $s' \rightsquigarrow \langle i, start, end \rangle$ if $s' = s_i[start, end]$
- the nodes can be represented by a perfect hash table which stores only the branching characters $\Rightarrow$ $O(1)$ branching time, space optimal

	Space
Trie (T)	$O(N+n) = O(N)$ (int. nodes, leaves)
Compacted trie (CT)	$O(n+n) = O(n)$ (int. nodes, leaves)

2-level indexing

1) Strings in Disk:


```
RETRIEVE1(Q, CT, D) // Q[1,p], D disk
    u ← DOWNWARD_TRAVERSAL(Q, CT); // u int. node s(u) = Q
    return LEAVES(SUBTREE(u, CT), D);
```

- Extra space: $O(\frac{n}{B})$
- Time: $O(p + u_{occ})$
- I/Os: $O(p + \frac{u_{occ}}{B})$ (*)

```
SEARCH(Q, CT, D) // Q[1,q]
    u, p ← DOWNWARD_TRAVERSAL(Q, CT); // s(u) = Q[1,p], p ≤ q
                                      (we can stop into an edge)
    j ← COMPARE(NEXT(s(u)), Q[p+1]); // S_j ≤ Q < S_{j+1}
    return SCAN(R_j); // return ...
```

- Extra space: $O(\frac{n}{B})$ ← the leaves
- Time: $O(p + FC_B(O_B))$ (CT, ...)

$\ldots$ (NEXT(S[m]), Q[p+1]); // $S_j \leq Q < S_{j+1}$
return SCAN(B_j); // return position of Q

RETRIEVE2(Q, (T, D) // Q[1, p]
  $\ell \leftarrow$ SEARCH(Q\$, CT, D);
  $r \leftarrow$ SEARCH (Q#, CT, D);
  return D[ℓ, r]; // all strings on disk prefixed by Q

- Time: $O(p + FC_B(D_B))$ — ↑ CT traversal, ↑ scan the block
- I/Os: $O(p + \frac{FC_B(D_B)}{B})$

- Extra space: $O(\frac{n}{B})$
- Time: $O(2p + \frac{2FC_B(D_B)}{B})$ — ↑ 2 SEARCH
- I/Os: $O(2p + \frac{2FC_B(D_B)}{B})$

(4)
Thm (RETRIEVE1) time is $O(p + n_{occ})$, #I/Os is $O(p + \frac{n_{occ}}{B})$ for prefix-search and retrieving strings prefixed by Q takes $O(FC_B(D_{occ}))$ time and $O(\frac{FC_B(D_{occ})}{B})$ I/Os.
The space is $O(\frac{n}{B})$ for the CT with 2-level indexing.

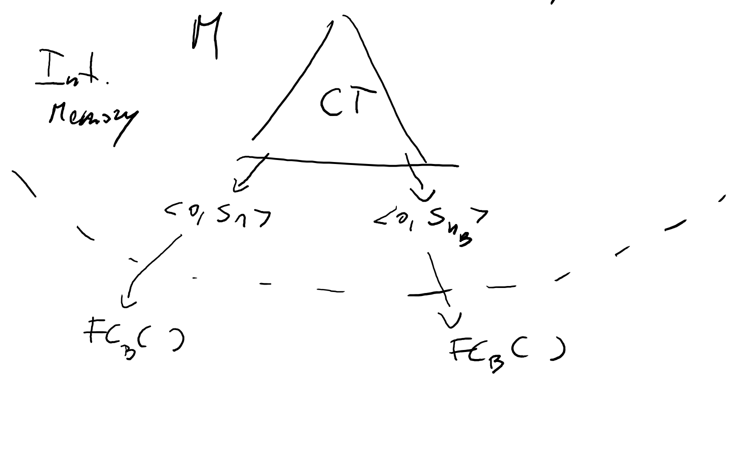
Proof time: $O(p + n_{occ})$
- p ↑ downward traversal
- n_{occ} ↑ # leaves descending by u (S[u] = Q)

I/Os: $O(p + \frac{n_{occ}}{B})$
- p ↑ no jump
- $\frac{n_{occ}}{B}$ ↑ on disk are contiguous and ordered

To retrieve all the occurrences we must display them, therefore we need $FC_B(D_{occ})$ time and $\frac{FC_B(D_{occ})}{B}$ I/Os
(using front coding, contiguous and ordered string on Disk)

∎

2) strings $S_1, \ldots, S_{n_B}$ on internal memory



SELECT(Q/CT, D)
// like SEARCH of 1)

RETRIEVE(Q, CT, D)
// like RETRIEVE2 of 1)

- time: $O(p + fC(B))$
- I/Os: $O(1)$ ← downward traversal compares with string in memory

---

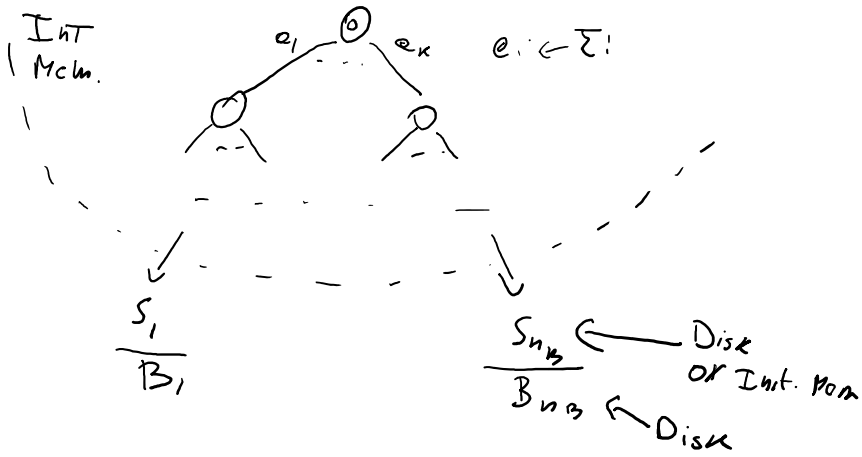
- time: $O(2p + 2fC(B))$
- I/Os: $O(1)$ (2 I/Os at most)
  ↖ 2 searches

Patricia trie

mercoledì 15 marzo 2023 17:04

Does not depends if strings are in IM or disk!

Def Patricia trie (PT) is a CT with an edge label only the first char. (≡ lossy CT)




```
RETRIEVE(Q, PT, D) // Q[1,p]
    l ← SEARCH(Q$, PT, D);
    r ← SEARCH(Q#, PT, D);
    return D[l,r];
```

- Space: $O(\frac{n}{B})$
- Time: $O(p)$
- I/Os: $O(1+\frac{p}{B})$

} th

```
SEARCH(Q, PT, D) // Q[1,q]
    s, p ← DOWNWARDTRAVERSAL(Q, PT) // s (arbitrary) leave of the subtree of u
                                    (u s.t. str(u) = Q[1,p], p ≤ q)
    mismatch ← LCP(s, Q) + 1;
    j ← UPWARD_TRAVERSAL(Q[mismatch], PT) // compare Q[mismatch] with s[u+1]
                                          // j is s.t. s_j < Q < s_{j+1}
    pos ← SCAN(Q, B_j);
    return pos;
```

Thm Prefix-search need $O(p)$ time and $O(1+\frac{p}{B})$ I/Os and $O(\frac{n}{B})$ space.

Proof time: $O(\underbrace{p}_{\text{downward traversal}} + \underbrace{p+|s|}_{LCP(s,Q)}) = O(p)$

I/Os: $\underbrace{0}\ I/Os + \underbrace{1}\ I/Os + \frac{p}{B}\ I/Os = O(\frac{p}{B}+1)\ I/Os$

$\overline{I/O_s}$: $\underbrace{0}_{\text{dummy internal (PT in int. node)}}$ $\overline{I/O_s}$ + $\underbrace{1}_{\text{pick the leaf (jump to disk)}}$ $\overline{I/O_s}$ + $\frac{p}{B}$ $\overline{I/O_s}$ = $O(\frac{p}{B}+1)$ $\overline{I/O_s}$ ← LCP

space: $O(\frac{n}{B})$ leaves, boundary interval node $\Rightarrow O(\frac{n}{B})$ int. node

$\Rightarrow O(\frac{n}{B})$ total space.

Given text string $T[1,n]$, alphabet Σ s.t. $|\Sigma| = \sigma$ and $T[n] = \$$ ($\$ \neq \forall c \in \Sigma$)
We want to resolve 2 ops:

- Counting ($P[1,p]$) = # occ of $P[1,p]$ as substring of $T[1,n]$
- Retrieve ($P[1,p]$) = $\{i \in \{1,\dots,n\} \mid T[i, i+p-1] = P[1,p]\}$

Def $Suff_i := T[i,n]$, $SUF(T) = \{Suff_i(T) \mid i = 1,\dots,n\}$

oss $P[1,p]$ substring of $T[1,n]$ $\iff$ $P[1,p]$ prefix of $s \in SUFF(T)$

Proof $P[1,p]$ substring of $T[1,n]$ $\iff$ $T[i, i+p-1] = P[1,p]$
$\iff$ $\exists i$ s.t. $P[1,p]$ prefix of $Suff_i$
$\iff$ $P[1,p]$ prefix of $s \in SUFF(T)$. ▷

Suffix array

giovedì 16 marzo 2023 01:17

$T[1,n]$, $SA[1,n] := [i_1, \dots, i_n]$ s.t. $Suff_{i_1}(T) < \dots < Suff_{i_n}(T)$

```
// BINARY SEARCH of P as prefix of suff(T)
SUBSTRING_SEARCH( P, SA(T) ) // P[1,p]
    l ← 1; r ← n.
    while (l ≠ r) {
        m ← ⌊(l+r)/2⌋.
        if( P > Suff_SA[m][1,p] )
            l ← m+1.
        else
            r ← m
    }
    if (P = Suff_SA[l][1,p])
        return l.
    else
        return -1;
```

- Extra space: $O(n)$ and $O(n \lg n)$ bits (*)
- Time: $O(p \lg n)$
- I/Os: $O(\lg n)$

Lemma Given $T[1,n]$ e $SA[1,n]$, substring search of P in T needs $O(p \lg n)$ time, $O(\lg n)$ I/Os and $O(n(\lg n + \lg \sigma))$ bits for space ($O(n \lg n)$ extra space). Retrieving all the occurrences need $O(occ)$ time.

Prof time: $O(p \lg n)$ — p: string comparison; $\lg n$: binary search

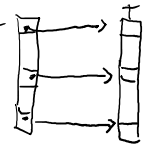
I/Os: $O(\lg n)$ jumping at most for BS

Total space: $O(n \lg \sigma + n \lg n)$ — $n \lg \sigma$: text; $n \lg n$: SA (extra space)

Retrieving: scan occ time ($\Rightarrow O(occ)$) starting from l obtained by SUBSTRING_SEARCH

Suffix array construction

C-like method T as array of char and SA as array of pointers to char

start: 

$\rightsquigarrow$ end: 


```
SA_BUILD(char* T, int n, char** SA) // C-like
    for(i = 0; i<n; i++)
        SA[i] = T+i; // pointer to T[i,n]
    QSORT(SA, n, sizeof(char*), SUFFIX_CMP);
```

- Working space: $O(n \lg n)$ bits
- Time: $O(n^2 \lg n)$
- I/Os: $O(\frac{n}{B} n \lg n)$

(*)

(+) using string pointers we don't need T's position in memory

(-) No locality in suffix comparison $\Rightarrow$ not efficient in I/Os

```
SUFFIX_CMP(char** p, char** q)
    // returns >0 if string p > string q
```

| ⇒ not efficient in I/Os

```
SUFFIX_CMP(char** p, char** q)
    return STR_CMP(*p, *q); // number > 0 if string p > string q
                            //        = 0 if          =
                            //        < 0 if          <
```

(*)

Thm Using a comparison-based sorter to build SA array need $O(n^2 \lg n)$ time $O(\frac{n}{B} n \lg n)$ I/Os and $O(n \lg n)$ working space.

Proof time: $O(n) + O(n) \cdot O(n \lg n) = O(n^2 \lg n)$

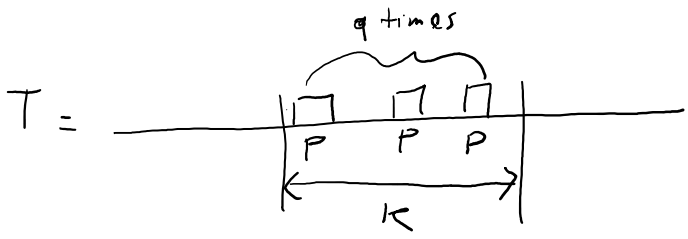
- $O(n)$: FOR
- $O(n)$: length of string $T[i, n]$
- $O(n \lg n)$: QSORT

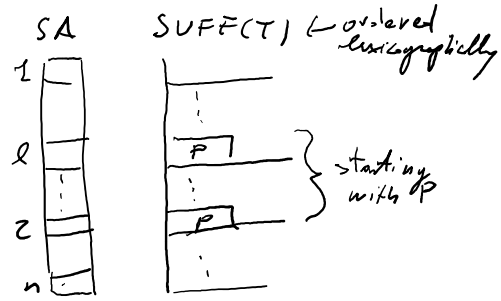
I/O: $O(\frac{n}{B} n \lg n)$

- $\frac{n}{B}$: string are on disk
- $n \lg n$: QSORT

Working space: $O(n \lg n)$ to store the n pointers

Example Given text $T[1..n]$, his SA array and a pattern $P[1,p]$, establish whether it exists a substring of T of length k which includes at least q occurrences of P






```
CHECK(T, SA, P, k, q)
    l,r ← SEARCH(P, SA); // binary search of P in SA
    occ ← r-l+1; // all occurrences of P in T
    A[1,occ] ← ZEROS(1,occ);
    for i = 1,...,occ
            A[i] ← SA[l+i-1]; // copy the starting pos. of P in T
    A ← SORT(A);
    i ← 1; res ← false;
    while(i < occ-q+1 AND NOT(res)){// search q occurrences of P that are k pos apart
            if(A[i+q-1]-A[i] ≤ k)
                    res ← true;
            i++;}
    return res;
```

Prop time complexity is $O(p \lg n + occ \lg occ)$

Proof time: $O(p \lg n) + O(occ) + O(occ \cdot \lg occ) + O(occ - q) = O(p \lg n + occ \lg occ)$

- $O(p \lg n)$: bin. search of P in SA
- $O(occ)$: copy SA[l,r] in $A[1,occ]$
- $O(occ \cdot \lg occ)$: sort $A[1,occ]$
- $O(occ - q)$: last for

Source $S \to s_1, s_2, s_3, \dots$ $(s_i \in \mathbb{N}^+)$, Alphabet Σ

Def Entropy of S: $H(S) := \sum_{s \in \Sigma} p(s) \cdot \underbrace{\lg_2 \frac{1}{p(s)}}_{\text{self information of } s} = E[-\lg_2 p(S)]$ $\left(E[g(X)] = \sum_x g(x) \cdot p(x)\right)$

($H(S)$: RV; $p(S)$: RV)

oss $0 \le H \le \lg_2 |\Sigma|$, and H is max when p is uniform

Proof Using Lagrange multipliers with constraint $\sum_{s \in \Sigma} p(s) = 1$, we obtain that H is maximum when $p(s) = \frac{1}{|\Sigma|}$ (uniformly distributed).

if p is uniform $p(s) = \frac{1}{|S|}$ $H = E[-\lg_2 \frac{1}{|\Sigma|}] = E[\lg_2 |\Sigma|] = \lg_2 |\Sigma|$ (constant)

Def Given coding C. Average Lenght of codeword is

$$L_C := \sum_{s \in \Sigma} p(s) |cw(s)|, \quad cw(s) = \text{"codeword of } s\text{"}$$

$p(s)$ can be
- static (prob. distr. given)
- semi-static (we know the frequency $f(x)$ $\forall x \in S$)
- dynamic (we must compute the frequency dinamically)

Shannon thm for all prefix-free code C: $L_C \ge H$

oss1 C is optimal $\iff L_C = H \iff |cw(s)| = \lg_2 \left(\frac{1}{p(s)}\right)$ $\forall s \in \Sigma$

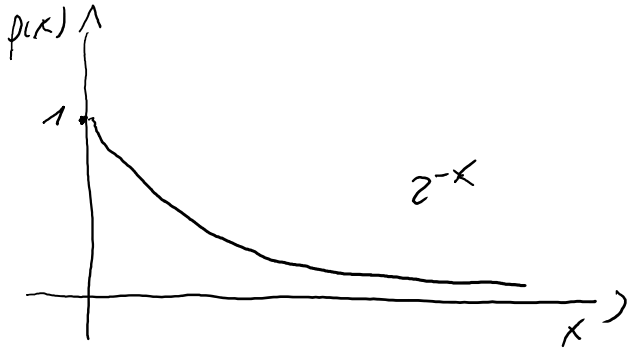
oss2 C is optimal for prob. dist. $p(x) = 2^{-|cw(x)|}$

Def A codeword C is universal iff $|c(x)| = O(\lg(x))$ $\forall x \in \mathbb{N}$

Def A standard C is universal iff $|C(x)| = O(g(x))\ \forall x \in \mathbb{Z}$

Unary code

giovedì 16 marzo 2023 17:24



Def for $x>0$ the unary code of x is

$$U(x) := 0^{x-1}1 .$$

Prop $|U(x)| = x$

Prop U is optimal for $p(x) = 2^{-x}$

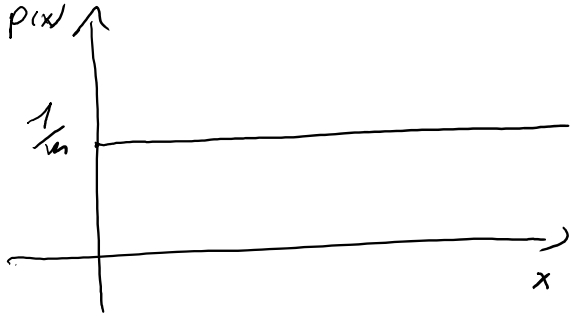
Proof $|cw(x)| = |U(x)| = x \Rightarrow$ optimal for $p(x) = 2^{-x}$

```
DECODE(T):
    next ← NEXTBIT(T);
    while (next ≠ EOF) {
        y ← 1;
        while (next ≠ 1) {
            y++;
            next ← NEXTBIT(T);
        }
        OUTPUT(y);
    }
    return;
```

Fixed-lenght binary code

giovedì 16 marzo 2023 17:34

Def Given $m = \max_{s_j \in S} \{s_j\}$ and $M = 1 + \lfloor \log_2 m \rfloor$, fixed-lenght binary code of x is $B_M(x)$.



Lemma $|B_m(x)| = M$

Prop It's optimal $\Leftrightarrow$ $p(x) \approx \frac{1}{m}$

Proof $p(x) = 2^{-M} \approx 2^{-\log m} = \frac{1}{m}$ □

```
DECODE(T)
i ← 1;
while(i < LEN(T)){
    x ← T[i, i+M];
    y ← DECIMAL(x, M); // from binary to decimal representation
    i ← i+M;
    OUTPUT(i);
}
return;
```

Gamma-code (Elias)

giovedì 16 marzo 2023 17:43

Def Given $x > 0$, $\gamma(x) := 0^{\ell-1} B(x)$, $\ell = |B(x)|$ $\quad (\gamma(x) = 0^{|B(x)|-1} B(x))$

Prop $|\gamma(x)| = 2\lfloor \log_2 x \rfloor + 1$

Proof $|\gamma(x)| = |B(x)| - 1 + |B(x)| = \lfloor \log_2 x \rfloor + \lfloor \log_2 x \rfloor + 1 = 2\lfloor \log_2 x \rfloor + 1$

Prop Optimal $\Leftrightarrow p(x) \approx \frac{1}{2x^2}$

Proof $p(x) = 2^{-|\gamma(x)|} = 2^{-2\lfloor \log_2(x) \rfloor - 1} \approx \frac{1}{2x^2}$

oss $\gamma(x)$ is universal prefix-free code

proof $|\gamma(x)| = 2\lfloor \log_2 x \rfloor + 1 = \mathcal{O}(\log x)$

```
DECODE(T)
next ← NEXTBIT(T); start ← 1;
while(next ≠ EOF){
        ℓ ← 0;
        while(next = 0)
            {ℓ++; next ← NEXTBIT(T);}
        start ← start + ℓ;
        X ← T[start, start++ℓ+1];
        y ← DECIMAL(X, ℓ+1);
        OUTPUT(y);
        start ← start + ℓ + 1;
```

```
        OUTPUT(y);
        start ← start + l + 1];
}
```

Delta-code (Elias)

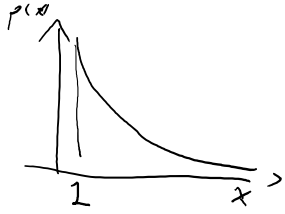
giovedì 16 marzo 2023 18:24

Def For $x \geq 0$ $\quad \delta(x) := \gamma(|B(x)|)\, B(x) \quad \left(= 0^{|B(|B(x)|)|-1} B(|B(x)|)\, B(x)\right)$

Prop $|\delta(x)| \approx 1 + \log_2 x + 2\log_2(\log_2 x)$ bits

Proof $|\delta(x)| = \underbrace{\lfloor \log_2(x) \rfloor + 1}_{B(x)} + \underbrace{2\lfloor \log_2(\overbrace{\lfloor \log_2 x \rfloor + 1}^{B(x)}) \rfloor + 1}_{\gamma(|B(x)|)} \approx 1 + \log_2 x + 2\log_2(\log_2 x)$ ☐

Prop $\delta(x)$ optimal $\iff p(x) \approx \frac{1}{2x(\log_2 x)^2}$



Proof $2^{-|\delta(x)|} \approx 2^{-(1+\log_2 x + 2\log_2(\log x))} = \frac{1}{2\, 2^{\log_2 x} \cdot 2^{\log_2(\log_2 x)^2}} = \frac{1}{2x(\log_2 x)^2}$ ☐

Prop $\delta(x)$ is universal

Proof $|\delta(x)| = 1 + \log_2(x) + 2\log_2\log_2(x) = \Theta(\log_2(x)) + o(\log_2(x)) = O(\log_2(x))$

```
DECODE_DELTA(T)
    next ← NEXTBIT(T); start ← 1;
    while (next ≠ EOF) {
        l ← 0;
        while (next = 0) {
            l++;
            next ← NEXTBIT(T);
        } start ← start + l;
```

```
} start←start+l.
L ← DECIMAL ( T[ start, start+l+1 ]);
start←start+l+1.
y ← DECIMAL (T[ start, start+L ]);
OUTPUT(y);
```

Rice code

giovedì 16 marzo 2023 18:54

Def Given $x > 0$ and a parameter $k \in \mathbb{Z}, k > 0$

$$R_k(x) := U(q+1)\, B_k(r)$$
$$(x-1 = q \cdot 2^k + r)$$

(quotient $q = \left\lfloor \frac{x-1}{2^k} \right\rfloor$, remainder $r = x-1-q \cdot 2^k$)

If $x \geq 0$ (CASE)

$$R_k(x) = U(q+1)\, B_k(r)$$
$$(x = q \cdot 2^k + r)$$

Prop $|R_k(x)| = \left\lfloor \frac{x-1}{2^k} \right\rfloor + 1 + k$ bits

Proof $|R_k(x)| = \underbrace{q+1}_{U(q+1)} + \underbrace{k}_{B_k(r)} = \left\lfloor \frac{x-1}{2^k} \right\rfloor + 1 + k$ ∎

Prop Optimal iff $p(x) = (1-p)^{x-1} p$ (geometric distribution $p(G=x)$) with parameter p of success

(NO PROOF)

Variabile-byte code

giovedì 16 marzo 2023 19:16

Def $x \geq 0$ $B(x) = B_k(x) B_7(x) \dots B_7(x)$, s.t. $7 + \dots + 7 + k = \lfloor \log_2(x) \rfloor + 1$

$VB(x) := 1\, B_7(x)\, 1\, B_7(x) \dots 1\, B_7(x)\, 0\, B_7(x)$

$B_k(x)$ padded with 0 to left

Prop $|VB(x)| = 8 \cdot \lceil \frac{|B(x)|}{7} \rceil$ bits.

Proof $B(x)$ is divided in block of 7 bits (the leftmost is padded) ($\lceil \frac{|B(x)|}{7} \rceil$ blocks) and for each block is added 1 or 0 $\Rightarrow$ 8 bit per each block ☐

Prop $VB(x)$ optimal iff $p(x) \propto \sqrt[7]{\frac{1}{x^8}}$

Proof $2^{-|VB(x)|} = 2^{-8\lceil \frac{|B(x)|}{7} \rceil} = 2^{-\frac{8}{7}(\lfloor \log_2(x) \rfloor + 1)} \approx \sqrt[7]{\frac{1}{x^8}}$ ☐

Analysis es (S,C)-dense code

(S,C)-Code with $S + C = 2^b$, $b = 8$

$S = \{0 b_1 \dots b_7 \mid b_i \in \{0,1\}\} = |\{0,1,\dots,127\}| = 128$

stoppers

$C = |\{1 b_1 \dots b_7 \mid b_i \in \{0,1\}\}| = |\{128,\dots,255\}| = 128$

$128 + 128 = 2^8 = 256$

	# representable	set
S	128	$\{0, \ldots, 127\}$
CS	$128 \cdot 128$	$\{128, \ldots, 128 + 128^2 - 1\}$
CCS	$128 \cdot 128 \cdot 128$	$\{128 + 128^2, \ldots, 128 + 128^2 + 128^3 - 1\}$

etc...

(s,c)-dense code

venerdì 17 marzo 2023 01:06

$s+c=2^b$, $\begin{cases} s = |\text{Stoppers}| \\ c = |\text{Continuers}| \end{cases}$

$\text{Stoppers} = \{0 \le B_b(x) = x_1 \dots x_b < s\}_{x_i \in \{0,1\}}$

$\text{Continuers} = \{s \le B_b(x) = x_1 \dots x_b < 2^b\}_{x_i \in \{0,1\}}$

Def $DC_{(s,c)}(x) := c_1 \dots c_k s$, $c_i \in \text{Continuers}$, $s \in \text{Stoppers}$

$DC_{(s,c)}$	# representable	Set
s	s	$\{0, \dots, s-1\}$
$c_1 s$	$c \cdot s$	$\{s, \dots, s+c \cdot s - 1\}$
$c_1 c_2 s$	$c^2 \cdot s$	$\{s+c \cdot s, \dots, s+c \cdot s+c^2 \cdot s - 1\}$

PForDelta code

venerdì 17 marzo 2023 01:18

$S = s_1, \ldots, s_n, \ldots$ and $base := \min_{s_i \in S} \{s_i\}$

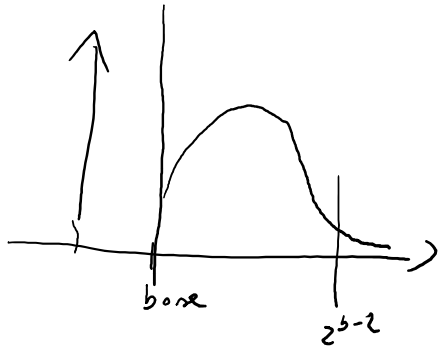
Choose b s.t. most (~90%) of $s_i \in \{base, \ldots, base + 2^b - 2\}$

Def $PFD(x - base) = \begin{cases} B_b(x - base) & \text{if } x - base \in \{0, \ldots, 2^b - 2\} \\ Esc & \text{if } x - base \geq 2^b - 1 \end{cases}$

$Esc \rightarrow B_{b+w}(x - base)$ ← fixed-size representation

Prop $|PFD(x - base)| = \begin{cases} b \text{ bits} & \text{if } x - base \in \{0, \ldots, 2^b - 2\} \\ b + w \text{ bits} & \text{if } x - base \geq 2^b - 1 \end{cases}$

oss Good for Gaussian-like distribution



Interpolative code

venerdì 17 marzo 2023 01:51

- Adaptive (depends on distribution of the seq.)
- NO prefix-free

$S = s_1, \dots, s_n$ $\quad \forall i \overset{s.p.g.}{\;} s_i < s_{i+1}$ $\quad$ (otherwise S' s.t. $s_i' = \sum_{j=1}^{i} s_j$)

// Start: $l = 1, r = n, low = s_1, hi = s_n$

```
IC(S, l, r, low, hi) // interpolative-coding
    if r < l
        return ε; // empty string
    if l = r
        return BC(S[l], low, hi);
    m ← ⌊(l+r)/2⌋;
    A1 ← BC(S[m], low + m - l, hi - r + m); // low + m - l ≤ S[m] ≤ hi - r + m
    A2 ← IC(S, l, m-1, low, S[m]-1);
    A3 ← IC(S, m+1, r, S[m]+1, hi);
    return [A1, A2, A3];
```

better low and hi

if equality $S[i] = S[l] + (i - l)$ $\forall i = l+1, \dots, r$

(+) Very compact
(-) very slow
(-) recursive (no cache friendly)
(-) $s_{i+1} > s_i$ $\forall i$

```
BC(x, a, b) // "succinct" binary code
    return B(x - a), ⌈log2(b - a + 1)⌉;
```

Elias-Fano code

venerdì 17 marzo 2023 16:48

$S = s_1, \dots, s_n \qquad s_i < s_{i+1}$

Let $\begin{cases} u := (\max S) + 1 = s_n + 1 \\ b := \lceil \lg_2(u) \rceil \\ \ell := \lfloor \lg_2(\frac{u}{n}) \rfloor \\ h := b - \ell \end{cases}$

$\forall x \in S \quad x \rightsquigarrow B_b(x) = \underbrace{H(x)}_{h} \underbrace{L(x)}_{\ell}$

Def Given $S = s_1, \dots, s_n$ and capital u, b, h and ℓ $\quad EF(S) := (L, H)$

where $\begin{cases} L := L(s_1) \dots L(s_n) \\ H := B_0 \dots B_{2^h-1}, \quad B_j := 1^{x_j} 0 \quad (x_j := |\{ H(s_i) = j \}|) \end{cases}$

($B_0, \dots, B_{2^h-1}$: buckets; $j = 0, \dots, 2^h - 1$)

Prop Given S with $|S| = n$, $|EF(S)| \le 2n + n\lceil \lg_2 \frac{u}{n} \rceil$ bits, so for $x \in S$ $|EF(x)| \le 2 + \lceil \lg \frac{u}{n} \rceil$ bits on avg (regardless the distribution of S).

Proof $|S| = n \quad |EF(S)| = |L| + |H| = \underbrace{n\ell}_{L} + \underbrace{\#1 + \#0}_{H} = n\lceil \lg_2(\frac{u}{n}) \rceil + \underbrace{n}_{\#1 = \#S} + \underbrace{2^h}_{\#0 = \#\text{buckets}}$

$= n\lceil \lg_2(\frac{u}{n}) \rceil + n + \underbrace{2^{\lceil \lg_2 u \rceil - \lceil \lg_2(\frac{u}{n}) \rceil}}$

$\le n\lceil \lg_2(\frac{u}{n}) \rceil + n + n$.

For $x \in S \quad |EF(x)| = \frac{|EF(S)|}{n}$ on avg $\le 2 + \lceil \lg_2(\frac{u}{n}) \rceil$

ass If S is uniformly distributed in $[0, u)$, EF requires less than 2 bits in addition by optimal per integer.

Proof S unif. distr. in $[0, u) \Rightarrow$ On avg $x \in S \; |C(x)| = \lceil \lg_2(\frac{u}{n}) \rceil$ (avg x) where C is optimal

∎

Augmenting the Elias-Fano coding (using RANK/SELECT data structure) we can perform two operations:

- $Access(i) = S_i \quad \forall i = 1, \dots, n$
- $NextGEQ(x) = \min\{S_i \mid S_i \geq x\} \quad \forall x = 0, \dots, u$

the auxiliary RANK/SELECT data structure support efficiently the RANK or SELECT ops

($x \in \{0,1\}$, B binary vector
- $Rank_x(i, B) = \#x$ in $B[1, i] \quad \forall i \leq |B|$
- $Select_x(i, B) =$ position of i-th x in B)

```
ACCESS(i, L, H, b):
    return Bin(SELECT_1(i, H) - i) L[(i-1)l+1, il];
```

to find the number of 0 before the i-th 1 $\Rightarrow$ to find the bucket

- Extra space: $o(|H|)$ bits $= o(n)$ bits
- Time: $O(1)$

$\}(*)_1$

```
NEXTGEQ(x, H, L):
    H(x), L(x) ← B_b(x); // H(x) first b bits, L(x) last l bits
    p ← SELECT_0(H(x), H) + 1;
    i ← p - H(x); // SELECT_0(H(x), H) - H(x) gives # of elements
                  // of the first H(x) buckets, so adding 1
                  // we have the position i of the first element of
    repeat {      //   (H(x)+1)-th bucket
        y ← ACCESS(i);   // (j+1)-th bucket = B_l(j)
        i++;
    } until (y ≥ x) // return first value ≥ x in S
    return y;
```

- Extra space: $o(|H|) = o(n)$ bits
- Time: observed $O(1)$
  worst case $O(\frac{u}{n})$
  worst case + binary search instead of scan (for) $O(\lg(\frac{u}{n}))$

$\}(*)_2$

Thm $(*)_1$ Using auxiliary $select_1$ data structure on H, ACCESS need $o(n)$ extra space and $O(1)$ time

Proof For this goal we use an auxiliary $select_1$ data structure on H that occupy $o(|H|)$ extra space.

$|H| = \#1 + \#0 = n + 2^{\ell} = n + 2^{\lfloor \lg_2 u \rfloor - \lfloor \lg_2 \frac{u}{n} \rfloor} \approx n + n = 2n = O(n)$

$\Rightarrow o(|H|) = o(n)$ extra space.

the time is $O(1)$ exploiting aux. $select_1$ data structure

(*)$_2$ Thm Using auxiliary Select$_0$ data structure on H, NEXTGEQ needs $o(n)$ extra space and $O(1)$ time on avg and $O(\frac{u}{n})$ in worst case ($O(\lg(\frac{u}{n}))$ in worst case using binary search in bucket)

Proof Space: using auxiliary select$_0$ d.s. that occupy $o(|H|) = o(n)$ extra space

Time: perform at most size(bucket(x)) times access operations (cycle FOR).

So on avg. $\text{size}(\text{bucket}(x)) = \frac{\#\text{buckets}}{n} = \frac{\#0}{n} = \frac{2^h}{n} \approx \frac{2n}{n} = O(1)$.

In the worst case the $\text{size}(\text{bucket}) = 2^\ell = 2^{\lceil \lg \frac{u}{n} \rceil} = O(\frac{u}{n})$

↑ after 2^ℓ we have a new bucket

Using binary search in the bucket we have $O(\lg \frac{u}{n})$

Rank and Select

lunedì 13 febbraio 2023 20:09

$B[1,m]$ binary array $\in \{0,1\}^m$

Def $Rank_1(B,k) := \sum_{i=1}^{k} B[i]$ $\quad (Rank_0(B,k) := k - Rank_1(B,k))$

Def $Select_1(B,k)$:= position of the k-th occurrence of 1 $\quad (Select_0(B,k) = \text{pos. of } k\text{-th } 0 \text{ in } B)$

Thm $\exists$ Rank data structure and Select data structure that takes $O(|B|) = O(m)$ bits and support respectively Rank and Select operation in $O(1)$ time

Proof See for rank succint rank implementation (Rank data structure)
For Select similar to rank (Select data structure) .

Succint solution for Rank (via Rank data structure)

giovedì 23 marzo 2023 02:35

$B[1,n]$ binary array

- Split $B[1,n]$ in $\frac{n}{z}$ block of size z:

$$B_i = B[z(i-1)+1, zi] \quad i = 1, \dots, \frac{n}{z}$$
$$|B_i| = z$$

For each B_i we store the absolute rank r_i (#1 from start to end of B_i)

$$\begin{cases} r_0 = 0 \\ r_i = Rank_1(B, zi) + r_0 \quad \text{if } i = 1, \dots, \frac{n}{z} \end{cases}$$

- Split each B_i in $\frac{z}{w}$ block of size w:

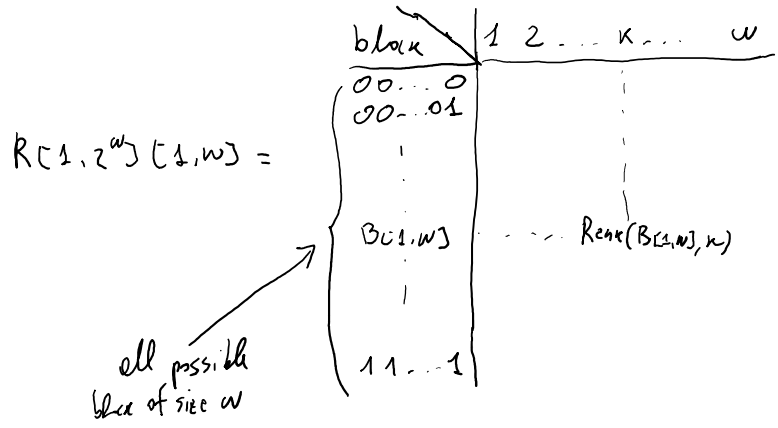
$$B_{i,j} = B[z(i-1) + w\cdot(j-1)+1, z\cdot(i-1) + w\cdot j] \quad j = 1, \dots, \frac{z}{w}$$
$$|B_{i,j}| = w$$

For each B_{ij} we store the relative rank r_{ij} (#1 from the start of B_{i1} to end of B_{ij})

$$r_{ij} = Rank_1(B_{i,1}, w) + \dots + Rank_1(B_{i,j}, w)$$
$$i = 1, \dots, \frac{n}{z}$$
$$j = 1, \dots, w$$

Four Russians trick (Rank data structure)

- Construct Look-up table R of size 2^w rows $\times$ w columns



Def the look-up table R for all possible binary of size w is the Rank data structure.

Oss Given a particular $B[1,w]$ block of size w, we have

$$R[B[1,w]][k] = \text{Rank}(B[1,w], k) \quad \text{for } k = 1, \dots, w$$

Succint implementation of Rank

```
// R = rank data structure, B[1,n], x index 1 ≤ x ≤ n
RANK(B, x)
    B_{i,j} ← SEARCH(x, B); // find the block where the index x is in
                               (z(i-1) + w·j(-1) + 1 ≤ x ≤ z(i-1) + w·j)
    k ← RELATIVE_POSITION(x, B_{i,j}); // position of index x in B_{i,j}
                                          (start counting from 1)
    return r_{i-1} + r_{i,j-1} + R[B_{i,j}, k];
```

- Extra space: $o(n)$ bits
- Time: $O(1)$ (*)

(*)

Thm Extra space for succint Rank is $o(n)$ bits using $z = \lg^2 n$ and $w = \frac{1}{2}\lg n$.
Time needed exploring the Rank d.s. is $O(1)$.

Proof Extra space = Space($\{r_i\}$) + space($\{r_{i,j}\}$) + space(R) (in bits)

$= O\left(\frac{n}{z}\lg n\right) + O\left(\left(\frac{n}{z}\cdot\frac{z}{w}\right)\lg z\right) + \text{space}(R)$

($\#r_i$, each r_i; $\#r_{i,j}$, each $r_{i,j}$; R = RANK data structure)

$= O\left(\frac{n}{z}\left(\lg n + \frac{z}{w}\lg z\right)\right) = O\left(\frac{n}{\lg^2 n}\cdot\left(\lg n + \frac{\lg^2 n}{\frac{1}{2}\lg n}\lg(\lg^2 n)\right)\right)$

$= O\left(\frac{n}{\lg^2 n}\cdot\left(\lg n + 2\lg n\cdot\lg(\lg^2 n)\right)\right) = O\left(\frac{n}{\lg^2 n}\cdot\left(\lg n + 4\lg n\cdot\lg(\lg n)\right)\right)$

$= O\left(\frac{n}{\lg n}\cdot\left(1 + 4\lg(\lg n)\right)\right) = O\left(\frac{n\lg(\lg n)}{\lg n}\right) = o(n) + \text{Space}(R)$

Space(R) = # rows · # columns · # bits per Rank (bits per all)

$= 2^w \cdot w \cdot \lg w = \sqrt{n}\cdot\frac{\lg n}{2}\cdot\lg\left(\frac{\lg n}{2}\right) =$

$= O(\sqrt{n} \cdot \lg n \cdot \lg(\lg n)) = o(n)$

$\Rightarrow$ Extra space $= o(n) + o(n) = o(n)$

Time: Look-up table + sum of values ($r_i, r_{i,j}$ + cell in R)

$\Rightarrow O(1)$

Corollary Total space is $n + o(n)$ bits

Proof total space $|B| + o(|B|)$

Compressed Rank/Select based on Elias-Fano coding

giovedì 23 marzo 2023 02:35

B binary array $B = [1, m]$ $|B| = m$, $n = \#1$ in B $\left(n = \text{Rank}(B, m)\right)$

$\delta = \frac{n}{m}$ density

Fact Using Rank data structure and Select data structure respectively for Rank and Select operations we need

- Total Space: $|B| + o(|B|) = m + o(m)$ bits
- Time: $O(1)$ for Rank or Select data structure

(-) If B is sparse ($\delta = \frac{n}{m} \approx 0$) we can do better.

Smart solution via EF-code

$B[1, m] \xrightarrow{\text{transform}} A[1, n]$, $n = \#1$ in B $\qquad A[i] = \text{SELECT}_1(B, i)$

```
CONSTRUCT_A(B) // B[1, m]
    for (i = 1, ..., n)
        A[i] = SELECT_1(B, i);
```

$\text{SELECT}_1(i, B)$

return ACCESS(i, A); // $A[i]$

- Total space: $2n + n\lceil \lg_2 \frac{m}{n} \rceil + o(n)$ bits
- Time: $O(1)$

$\Big\}$ $(*)_2$

$\text{RANK}_1(i, B)$

return INDEX(NEXTGEQ$(i+1, A), A) - 1$;

- Total space: $2n + n\lceil \lg_2 \frac{m}{n} \rceil + o(n)$ bits
- Time: $O(\lg \frac{m}{n})$

$\Big\}$ $(\beta)_2$

$(*)$ **Thm** Transform B in A and applying EF-coding to B we perform $\text{SELECT}_1(i, B)$ in $O(1)$ time taking $2n + n\lceil \lg_2 \frac{m}{n} \rceil + o(n)$ bits

Proof Time: $O(1)$ time for ACCESS(i, A) exploiting Select_1 data structure for H $\left(EF(A) = (L, H)\right)$ and select value on L (see ACCESS)

Total space: $|EF(A)| + (\text{Select}_1 \text{ d.s.})$

$$= |L| + |H| + (\text{Select}_1 \text{ d.c.}) =$$

$$\leq \underbrace{n\lceil \lg_2 \tfrac{m}{n} \rceil}^{(n \cdot \ell)} + \underbrace{2n}_{|H|} + \underbrace{o(n)}_{o(|H|)} \quad \text{bits}$$

$|L|$

□

Thm (*)$_2$ As before $\text{RANK}_1(i, B)$ takes $O(\lg \frac{m}{n})$ time exploiting Select_0 and Select_1 data structure build on H. The total space needed is $2n + n\lceil \lg_2 \frac{m}{n} \rceil + o(n)$ bits

Proof • Time is given by $\text{NEXTGEQ}(i+1, A)$, so in worst case $O(\lg \frac{u}{n})$, exploiting Select_0 and Select_1 data structure build on H ($u = \max A + 1$ see EF-code).

$u = \max A + 1 \leq m + 1 \Rightarrow$ in worst case $O(\lg \frac{m}{n})$.

• Total space: $|EF(A)| + \text{Select}_0 \text{ for } H + \text{Select}_1 \text{ for } H$

$$\leq \underbrace{n\lceil \lg_2 \tfrac{m}{n} \rceil}_{L} + \underbrace{2n}_{H} + o(n)$$

Analysis

	R/S data structure		Via Elias-Fano Code		
	Time	Space		Time	Space
Rank	$O(1)$	$\mathcal{O}(m) + o(m)$	$\equiv \text{NextGEQ}$	$O(\lg_2 \frac{m}{n})$	$2n + n\lceil \lg_2 \frac{m}{n} \rceil + o(n)$
Select	$O(1)$	$\mathcal{O}(m) + o(m)$	$\equiv \text{Access}$	$O(1)$	$2n + n\lceil \lg_2 \frac{m}{n} \rceil + o(n)$

• R/S data structure on B vs. EF-code on A w.r.t. density of B:

	B dense ($\frac{m}{n} \approx 1$)				B sparse ($\frac{m}{n} \ll 1$)			
	R/S data		EF-code		R/S data		EF-code	
	Time	Space	Time	Space	Time	Space	time	space
Rank	+	+	~	~	+	—	—	+
Select	≈	+	≈	~	≈	—	≈	+

Succint representation of binary tree (LOUDS-like)

giovedì 23 marzo 2023 02:37

Binary tree T, node $x \in T$ with label l

Def x is a binary node $\iff$ has two child

```
ENCODE(T)
    T' ← COMPLETE_WITH_DUMMY_NODES(T); // complete all non-binary nodes with dummy leaves
    T' ← ENUMERATE_NODES(T'); // left to right and downward (all node)
    for i = 1,...,2n+1 {
        if T'[i] ≠ dummy // i-th node
            B[i] = 1;
        else
            B[i] = 0;
    }
    T' ← ENUMERATE_NO DUMMY(T');
    for i = 1,...,n
        L[i] = LABEL(T'[I[i]]); // T'[x][y]  (T'[I[i]] = T[i])
                                //  ↑ history  ↖ 2nd enum.
```

Def A binary tree is full $\iff$ each node has 0 or 2 children

Oss T' is a full binary node

Proof each original node has 2 children, leaves (dummy node) have 0 children ☐

Oss B binary array $|B| = 2n+1$ $(B[1, 2n+1])$
L label array $|L| = n$ $(L[1, n])$

Proof $|B| = m = \#$ nodes of $T' = \#$ internal nodes $+ \#$ leaves $= \#$ original nodes $+ \#$ dummy nodes
$= n + (n+1) = 2n+1$
T' is a full binary tree ☐

Oss1 $T[Rank_1(B, i)] = T'[i]$

Oss2 Given a node i in T $(T[i])$ $T'[2i]$ is the left child in T' and $T'[2i+1]$ is the right child in T'. (T' complete)

Oss3 $T'[Select_1(B, i)] = T[i]$

Oss4 Given a node i in T' $(T'[i])$ $T[\lfloor \frac{i}{2} \rfloor]$ is the parent in T.

OSS4 Given a node i in T' ($T'[i]$) $T[\lfloor \frac{i}{2} \rfloor]$ is the parent in T.

```
//Input: i = i-th node of T
LEFTCHILD(i) // using OSS1 and OSS2
    if B[2i] = 1
        return RANK_1(B, 2i);

//Input: i = i-th node of T
RIGHTCHILD(i) // using OSS1 and OSS2
    if B[2i+1] = 1
        return RANK_1(B, 2i+1);

//Input: i = i-th node of T
PARENT(i) // using OSS3 and OSS4
    j ← ⌊SELECT_1(i, B) / 2⌋;
    if B[j] = 1
        return j;
```

- Space: $2n+1+o(n)$ bits
- Time: $O(1)$

(A)

Thm A binary tree of n nodes can represented in $2n+1+o(n)$ bits and supports PARENT, LEFTCHILD, RIGHTCHILD in $O(1)$ time. (without counting array of labels L)

Proof Space: $2n+1 + o(n)$ bits
$|B|$ R/S data structure for B ($o(|B|)$)

Time: PARENT, LEFTCHILD, RIGHTCHILD $O(1)$ using rank or select and exploiting R/S data structure.

General

lunedì 13 febbraio 2023 20:07

(Text conversion)

Alphabet $\Sigma = \{\text{characters}\}$, $\sigma = |\Sigma|$

↑ atomic items

Huffman coding

sabato 18 marzo 2023 00:19

- Prefix-free code

$\forall s \in \Sigma'$, $p(s)$ given $\sum_{s \in \Sigma'} p(s) = 1$

```
HUFFMANN_TREE(Σ'):
    candidates ← {Σ'}; // queue with priority p(s)
    while(|candidates|>1){
        u1, u2 ← pop(candidates, 2); // extract 2 items with minimum prob.
        u ← PARENT(u1, u2) // create parent with child u1 and u2 and label edges with 0 and 1
        ADD((u, p(u1)+p(u2)), candidates);
    }
```

(p(u) = prob of u)

- Space: $\Theta(|\Sigma'|)$ (*)
- ↳ space high if $|\Sigma'|$ high

(*) Prop The Huffmann tree takes $\Theta(|\Sigma'|)$ space

Proof Huffmann tree is a full binary tree (each node has 2 or 0 childs)

We prove that T full binary tree with n = # leaves $\Rightarrow$ # internal nodes = $n-1$:

$n=1$) $T=$ • 0 internal node ok.

$n \to n+1$) 

L, R two full subtree of the root. ($J(L)$ = # internal node of L, $J(R)$ = # // // of R)

L, R have respectively k_1 and k_2 leaves. $\Rightarrow J(L) = k_1 - 1$, $J(R) = k_2 - 1$ and $k_1 + k_2 = n+1$

$\Rightarrow J(T) = J(L) + J(R) + 1 = k_1 - 1 + k_2 - 1 + 1 = k_1 + k_2 - 1 = (n+1) - 1 = n$

Huffmann tree has $|\Sigma'|$ leaves $\Rightarrow |\Sigma'| - 1$ internal node $\Rightarrow |\Sigma'| + |\Sigma'| - 1 = \Theta(|\Sigma'|)$ space ∎

```
HUFFMANN_CODE(s ∈ Σ'):
    T ← HUFFMANN_TREE(Σ');
    LABELING(Σ');
    DOWNWARD_TRAVERSAL(T, s);
    // print the label of edges ∈ path (labels[0], ...)
```

↳ downward traversal $\Rightarrow$ (cache misses)

```
HUFFMAN_DECODE(B, T) // B[1,n], T Huffmann tree
    u ← T; // root
    i ← 1;
    while(i < n){
        while(u ∉ Σ'){
            u ← DOWNWARD(T, B[i]); // go through edge B[i]
            i++;
        }
        OUTPUT(u);
    }
```

Def Te is a binary tree that identify a code C by downward traversing. L_e is the average lenght of codeword and the average depth of the leaves of T_e

$$L_e = \sum_{s \in \Sigma'} p(s)\,|C(s)| = \sum_{s \in \text{leaves}(T)} p(s)\,\text{depth}(s)$$

Fact We can build different Huffman trees $T_1 \neq T_2$ obtaining different codes $H_1 \neq H_2$.
We'll have $L_{H_1} = L_{H_2}$ but $\max_{s \in \Sigma_1} |H_1(s)| \neq \max_{s \in \Sigma_1'} |H_2(s)|$

Aim $\min_H \left(\max_{s \in \Sigma_1} |H(s)| \right)$

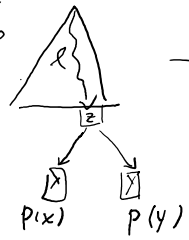
Strategy Among equal probability select the oldest node (the deepest ones)

Lemma 1 $\mathcal{F} = \{ T_e \mid L_{T_e} = \min_T L_T, T \text{ with } |\Sigma_1| \text{ leaves} \} \Rightarrow \exists T_e' \in \mathcal{F}$ s.t.

$T_e' =$ 

and $p(x), p(y)$ minimum prob. among $\{ p(s) \mid s \in \Sigma_1 \}$
$x, y \leftarrow$ max depth, $z = \text{parent}(x,y)$ (we can have other leaves at max depth)

Proof By contradiction Supposing that a leaf with minimum prob. is not at max depth.
By swapping it with a leaf at max depth but with higher prob, we obtain
a tree T with L_T lower $\lightning$

Lemma 2 Performing reduction on a tree T_e 

reduction → 

T_R (x, y with smallest prob.)

$\Rightarrow L_T = L_R + (p(x) + p(y))$

Proof $L_T = \sum_{s \text{ leaf}}' \text{depth}(s) \cdot p(s) = \sum_{\substack{s \neq x,y \\ s \text{ leaf}}}' \text{depth}(s) \cdot p(s) + \text{depth}(x) \cdot p(x) + \text{depth}(y) \cdot p(y)$

$= \sum_{s \neq x,y}' \text{depth}(s) \cdot p(s) + (l+1)(p(x) + p(y))$

$L_T = \sum_{s \text{ leaf}}' \text{depth}(s) \cdot p(s) = \sum_{s \neq z} \text{depth}(s) \cdot p(s) + \text{depth}(z) \cdot p(z)$

$$= \sum_{s \neq z} \text{depth}(s) \cdot p(s) + \ell \cdot (p(x)+p(y))$$

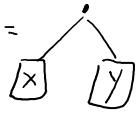
$$\Rightarrow L_T - L_R = p(x)+p(y)$$

■

Thm H Huffman code $\Rightarrow L_H \leq L_e$ $\forall$ prefix-free code e

(Equivalently: $L_{T_H} = \min_{T_e} \{L_{T_e}\}$)

Proof By induction on $|\Sigma'|$ (# leaves)

$|\Sigma'|=2$) $T_H =$ 

true (Huffman tree is full binary tree)

$|\Sigma'|-1 \to |\Sigma'|$) for lemma 1 $\exists T_e$ optimal with $|\Sigma|$ leaves s.t.

$T_e =$ 

$p(x), p(y)$ min. prob. and x, y at max depth with $z = \text{parent}(x,y)$

$\Rightarrow L_{T_e} \leq L_{T_H}$ given T_H Huffman tree of $|\Sigma'|$ leaves and for the characteristic of H, T_H is s.t. $= T_H$

← min prob.

Reducing T_e en T_H respectively in R_e and R_H, we have

$$L_{T_e} \overset{\text{Lemma 2}}{=} L_{R_e} + (p(x)+p(y)) \overset{\text{ind. hp.}}{\geq} L_{R_H} + (p(x)+p(y)) = L_{T_H}$$

L_{R_e}: # leaves $= |\Sigma|$; L_{R_H}: Huffman tree with # leaves $|\Sigma|$; L_{T_H}: Lemma 2

$\Rightarrow L_{T_H} \leq L_{T_e} \Rightarrow L_{T_H} = L_{T_e} \Rightarrow T_H$ is optimal

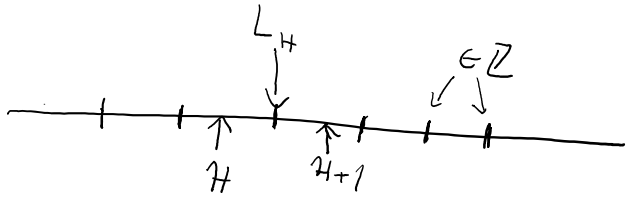
■

OSS e is Huffman $\overset{H}{\Rightarrow}$ e is optimal (L_e minimum)

oss C is Huffman $\Rightarrow$ C is optimal (L_C minimum)
$\not\Leftarrow$

Thm $S = s_1, \dots, s_n \Rightarrow H \le L_H < H+1$, where H is the entropy of S
(At most 1 bit lost for $cw(s)$ given $s \in S$)

Proof
$$L_H = \sum_{s \in \Sigma_1} p(s) \cdot |H(s)| = \sum_{s \in \Sigma_1} p(s) \cdot \underbrace{depth(s)}_{\in \mathbb{N}}$$
$$H = \sum_{s \in \Sigma_1} p(s) \underbrace{\log_2\left(\frac{1}{p(s)}\right)}_{\in \mathbb{R}}$$
} and H is optimal



oss The compress ratio is $\ge \frac{1}{\log_2 |\Sigma_1|}$

proof $|H(x)|$ is at least 1 bit for $x \in \Sigma_1$, and $\log_2 |\Sigma_1|$ is the longest number of bit for an item $\Rightarrow$ compression is $\ge \frac{1}{\log_2 |\Sigma_1|}$ ∎

Optimizations Consider $\Sigma_1^k = \{s_{i_1} \dots s_{i_k} \mid s_{i_j} \in \Sigma_1\}$ alphabet of blocks of symbols
$|\Sigma_1^k| = |\Sigma_1|^k$, $\forall x \in \Sigma_1^k$ $H(x)$ lost only one extra bit
$\Rightarrow H \le L_H < H + \frac{1}{k}$ (at most $\frac{1}{n}$ bit lost for each symbol)

Canonical Huffman coding

sabato 18 marzo 2023 16:50

- Implicit building of the Huffman tree to save space

```
// Start: num array, symb array of lists, fc array
IMPLICIT_CANONICAL_HUFFMANN_TREE(Σ, H) // H is an Huffmann code
    mex ← MAX(|H(s)|, s∈Σ); // max length of codewords
    for l = 1, ..., mex { // l = level = length of codewords (depth)
        symb[l,:] ← {s | |H(s)| = l}; // starting from index 0
                                      // lexicographic order
        num[l] ← LEN(symb[l,:]);
    }
    fc[mex] = 0
    for l = mex-1, ..., 1
        fc[l] ← (fc[l+1] + num[l+1]) / 2;
```

Space: $O(|\Sigma| + mex) \lg(|\Sigma|+1)$ bits (*)

```
CODE(s) // s∈Σ
    (i,j) ← SEARCH(s, symb); // symb[i,j] = s, i = level, j = j-th symbol of level i
    return B_i(fc[i] + j);
```

```
DECODE(C) C binary text
    v ← NEXT_BIT(C);
    while (v ≠ EOF) {
        l ← 1;
        while (v < fc[l]) { // when v ≥ fc[l], l is the level of v
            v ← 2v + NEXT_BIT(C);
            l++;
        }
        OUTPUT(symb[l, v - fc[l]]);
        v ← NEXT_BIT(C);
    }
    return;
```

(+) NO tree traversal ⇒ NO cache misses

(*) This Canonical Huffmann coding, not building the tree, needs $O(mex^2 + |\Sigma| \cdot (\lfloor \lg_2 |\Sigma| \rfloor + 1))$ bits, saving $O(|\Sigma|(\lfloor \lg_2 |\Sigma| \rfloor + 1))$ bits.

... storing $O(|\Sigma_i| \log_2 |\Sigma_i|)$ bits.

Proof Don't need to store num array but only fc array and sym table

$\Rightarrow$ Space: $O\left(\underbrace{\underbrace{\text{max} \cdot \text{max}}_{\text{max \# bits per item}}}_{\text{\# level}} + \underbrace{|\Sigma_i| \cdot \left(\lfloor \lg_2(|\Sigma_i|) \rfloor + 1\right)}_{\text{sym}}\right) = O\left(\text{max}^2 + |\Sigma_i| \cdot \left(\lfloor \lg_2(|\Sigma_i|) \rfloor + 1\right)\right)$ bits

fc

Space (Huffman tree) $= O\left(\underbrace{(|\Sigma_i| - 1)}_{\text{\# int. nodes}} \cdot \underbrace{2w}_{\substack{\text{\# pointer per int. node} \\ \text{\# bits per pointer}}} + |\Sigma_i| \cdot \lceil \lg_2(|\Sigma_i|) \rceil\right)$ bits

int. nodes

leaves

Arithmetic coding

sabato 18 marzo 2023 20:18

- Prefix code
- Compressed output is not a concatenation of codewords
- a bit of the output can represent more than one input symbol

bit stream $\beta = b_1 \dots b_k$ $\qquad \phi : \{0,1\}^* \longrightarrow [0,1) \subseteq \mathbb{R}$

$b_1 \dots b_k \longmapsto 0.b_1 \dots b_k = \sum_{i=1}^{k} b_i \cdot 2^{-i}$

Def Dyadic fraction $\frac{v}{2^n}$, $v, n \in \mathbb{N}^+$

Fact 1 $\forall\, b_1 \dots b_k$, $\phi(b_1 \dots b_k) = \frac{val(b_1 \dots b_k)}{2^k} \in [0,1)$

Proof $\frac{val(b_1 \dots b_k)}{2^k} = \frac{1}{2^k}\left(\sum_{i=1}^{k} b_i 2^{k-i}\right) = \sum_{i=1}^{k} b_i \cdot 2^{-i} = \phi(x)$ ∎

Fact 2 $\forall\, x \in [0,1)$ $\quad x = \frac{v}{2^n} \Rightarrow \phi^{-1}(x) = \beta_k(v)$

Proof As for Fact 1 ∎

Fact 3 $\forall x \in [0,1)$, $\phi^{-1}(x) = \text{CONVERTER}(x,k) = b_1 \dots b_k$ (*)

(*) CONVERTER (x,k) // $x \in [0,1)$, $k \in \mathbb{N}^+$ (print first k bits; possibly the stream has infinite length)

```
i ← 1;
while (i ≤ k) {
    x ← 2·x;
    if (x < 1)
        OUTPUT (0);
    else {
        OUTPUT (1);
        x ← x − 1;
    }
    i++;
}
```

Compression

$S = \sigma_1 \dots \sigma_n$, $\sigma_i \in \Sigma$ $\qquad p(\sigma_i)$ given $\forall i = 1, \dots, |\Sigma|$

$$f_{\sigma_i} := \begin{cases} 0 & \text{if } i=1 \\ f_{\sigma_{i-1}} + p(\sigma_{i-1}) & \text{if } 2 \le i \le n \end{cases}$$

```
CODE(S)
    (l_n, s_n) ← SEARCH_INTERVAL(S);
    (x, v, k) ← DYADIC([l_n, l_n + s_n)); // select x ∈ [l_n, l_n + s_n), x = v/2^k
    B ← B_k(v); // cw(S)
    return OUTPUT(B, n);

CODE_EFFICIENT(S)
    (l_n, s_n) ← SEARCH_INTERVAL(S);
    x ← l_n + s_n/2;
    B ← CONVERTER(x, ⌈lg_2 2/s_n⌉); // cw(S)
    return OUTPUT((B, n));

// p(σ_i), f_{σ_i} given and n = |S|
SEARCH_INTERVAL(S)
    s_0 ← 1; l_0 ← 0; // starting size and lower
    i ← 1;
    while (i ≤ n){
        s_i = s_{i-1} · p(σ_i); // new size
        l_i = l_{i-1} + s_{i-1} · f_{σ_i}; // new lower
        i++;
    }
    return (l_n, s_n); // interval is [l_n, l_n + s_n)
```

Decompression

(B, n) $B = b_1 \dots b_k$ $b_i \in \{0,1\}$, n = # items desired

```
// Start: p(σ_i), f_{σ_i} given
DECODE(B, n)
    x ← φ(B); // x = val(y)/2^u
    s_0 ← 1; l_0 ← 0;
    i ← 0;
    while (i < n){
        {l_i, l_i + s_i · f_{σ_1}, ..., l_i + s_i · f_{σ_n}} ← PARTITION([l_i, l_i + s_i));
```

```
{l_i, l_i + s_i·f_σ1, ..., l_i + s_i·f_σn} ← PARTITION([l_i, l_i + s_i));
j ← SEARCH(x, {l_i, ..., l_i + s_i·f_σn}); // x ∈ [l_i + f_σj·s_i, l_i + f_σj+1·s_i)
OUTPUT(σ_j);
s_i+1 ← s_i · p(σ_j);
l_i+1 ← l_i + s_i · f_σj;
}
return;
```

Analysis

Oss (compression) the size of the final interval does not depend on the particular ordering of input $S = \sigma_1 \dots \sigma_n$

Proof $s_n = s_{n-1} \cdot p(\sigma_n) = \dots = \underbrace{s_0}_{\uparrow 1} \cdot p(\sigma_1) \cdot \dots \cdot p(\sigma_n) = \prod_{i=1}^{n} p(\sigma_i)$

Def $x = 0.b_1 \dots b_n \in [0,1)$, $trunc_d(x) := 0.b_1 b_2 \dots b_d$ (for $1 \le d \le n$)

Lemma $\forall x \in [0,1)$, $trunc_d(x) \in [x - 2^{-d}, x]$

Proof $x \in [0,1) \Rightarrow x = \sum_{i=1}^{\infty} b_i 2^{-i}$, $trunc_d(x) = \sum_{i=1}^{d} b_i 2^{-i}$

$$\Rightarrow x - trunc_d(x) = \sum_{i=d+1}^{+\infty} b_i 2^{-i} = \sum_{i=1}^{+\infty} b_{d+i} 2^{-d-i} \le \sum_{i=1}^{+\infty} 2^{-d-i} = 2^{-d} \left(\sum_{i=1}^{+\infty} 2^{-i} \right)$$

($\uparrow$ $b_{d+i} = 1 \ \forall i$)

$$= 2^{-d} \cdot \left(\underbrace{\frac{1}{1 - \frac{1}{2}}}_{1} - 1 \right) = 2^{-d}$$

$$\Rightarrow x - 2^{-d} \le trunc_d(x) \le x$$

($\uparrow$ trivial)

Corollary $trunc_{\lceil \log_2 \frac{2}{s} \rceil}(l + \frac{s}{2}) \in [l, l+s)$ ← final interval

Proof By lemma $\begin{cases} trunc_{\lceil \log \frac{2}{s} \rceil}(l + \frac{s}{2}) \ge l + \frac{s}{2} - 2^{-\lceil \log \frac{2}{s} \rceil} \ge l + \frac{s}{2} - 2^{-\log \frac{2}{s}} = l \end{cases}$

Proof By lemma $\begin{cases} \text{Trunc}_{\lceil \lg \frac{2}{s} \rceil}(\ell + \frac{s}{2}) \geq \ell + \frac{s}{2} - 2^{-\lceil \lg \frac{2}{s} \rceil} \geq \ell + \frac{s}{2} - 2^{-\lg \frac{2}{s}} = \ell \\ \quad // \qquad\qquad \leq \ell + \frac{s}{2} \end{cases}$

$\Rightarrow \text{trunc}_{\lceil \lg_2 \frac{2}{s} \rceil}(\ell + \frac{s}{2}) \in [\ell, \ell + \frac{s}{2}] \subseteq [\ell, \ell + s)$ ∎

Thm $|AC(S)| \leq 2 + nH$, where $n = |S|$ and H is entropy of S

Proof $|AC(S)| = \lceil \lg_2 \frac{2}{s} \rceil = \lceil 1 + \lg_2 \frac{1}{s} \rceil \overset{(s = s_n)}{=} \lceil 1 - \lg_2 s_n \rceil < 2 - \lg_2 s_n$

$= 2 - \lg_2 \left(\prod_{i=1}^{n} p(\sigma_i) \right) = 2 - \sum_{i=1}^{n} \lg_2 p(\sigma_i) = 2 - \sum_{\sigma \in S} occ(\sigma) \lg_2 p(\sigma)$

($S = \sigma_1 \dots \sigma_n$, $s_0 = 1$)

$= 2 + \sum_{\sigma \in S} occ(\sigma) \lg \frac{1}{p(\sigma)} = 2 + n \sum_{\sigma \in S} \frac{occ(\sigma)}{n} \lg \frac{1}{p(\sigma)}$

$\overset{\wedge}{\approx} 2 + n \sum_{\sigma \in S} p(\sigma) \frac{1}{p(\sigma)} = 2 + nH$

($\frac{occ(\sigma)}{n} = freq(\sigma) \approx p(\sigma)$)

∎

Corollary $L_{AC} \leq H + \frac{2}{n}$ (L_{AC} = avg $|AC(\sigma)|$) (only $\frac{2}{n}$ bit wasted on avg. per symbol)

	Compression	On the fly	time	flexibility
Canonical Huffman	< (−)	no (−)	< (+)	yes (+)
Arithmetic Coding	> (+)	yes (+)	> (−)	no (−)

(yes: can use dynamic modeling)

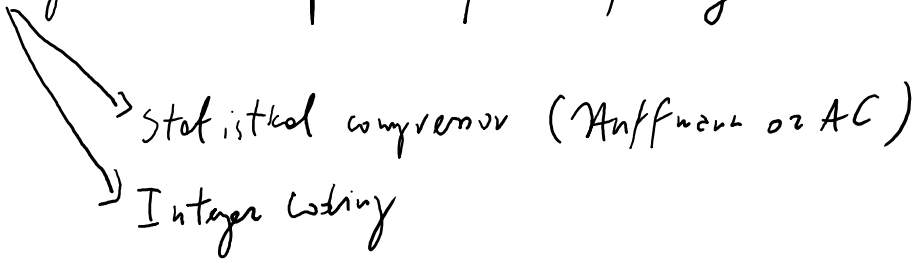
(yes: can decompress any portion provided its first codeword)

- No probabilities of the source S
- Given a dictionary of strings, we replace them with a token which identifies them in the dictionary

Compression

1) Parsing: input stream S ⟶ S' sequence of integers (atomic items)

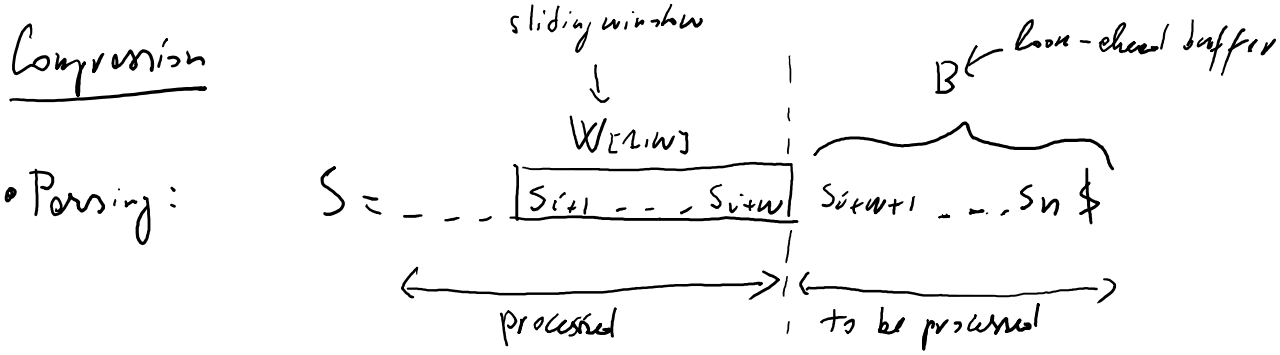
2) Encoding: encode the parsed sequence of integers S'



LZ77

lunedì 20 marzo 2023 14:08

Compression



$S \longrightarrow S' = \langle d_1, |\alpha_1|, c_1 \rangle, \ldots, \langle d_k, |\alpha_k|, c_k \rangle$

triplet $\in \mathbb{N} \times \mathbb{N} \times \Sigma$

- d_i = distance from start in W
- $|\alpha_i|$ = lenght of prefix
- c_i = next char in B

```
PARSING(S) //S = S_1 ... S_n
    W ← {}; B ← S; i ← 1;
    while (i ≤ n) {
            ⟨d, |α|, c⟩ ← PARSE(B, WB);
            i ← i + |α| + 1;
            W ← WB[i, n];
            B ← B[i, n];
            OUTPUT(⟨d, |α|, c⟩);
```

```
PARSE(WB, B) // the common prefix α in WB must be found in W
        d ← 0; |α| ← 0; c ← B[1]; i ← 0;
        while (i < |B|) {
                i++;
                if (LCP(W[|W|-i, |W|]B, B) ≥ |α|) {
                        |α| ← LCP(W[|W|-i, |W|]B, B);
                        c ← B[|α|+1];
                        d ← i;
                }
        }
```

}
}
return $\langle d, |\alpha|, c \rangle$;

- Encoding $S' = \langle d_1, |\alpha_1|, c_1 \rangle, \ldots, \langle d_k, |\alpha_k|, c_k \rangle$

Encode $\{d_1, \ldots, d_k\}$ (i.e. using Huffman)

Σ'	occ	freq $\approx p$
d_1	$\#d_1$	$\#d_1/k$
⋮	⋮	⋮
d_k	$\#d_k$	$\#d_k/k$

Output: $\langle H(d_1), |\alpha_1|, c_1 \rangle, \ldots, \langle H(d_k), |\alpha_k|, c_k \rangle$

OSS $w = |W| \uparrow \Rightarrow$ Compression $\uparrow$, time complexity $\uparrow$

$w = |W| \downarrow \Rightarrow$ // $\downarrow$, // $\downarrow$

Decompression

```
DECOMPRESS(S'): // S' = <d_1,|α_1|,c_1>,...,<d_k,|α_k|,c_k>
    S ← {}; i ← 1;
    for(i < |S'|){
        S ← [S, DECODE(S, <d_i,|α_i|,c_i>)];
        i ← i + |α_i| + 1; }
    return S;

DECODE(S, <d,|α|,c>)
    for i = 0,...,|α|-1 // out = zeros[1,|α|]
        out[i+1] ← S[|S|-d+1];
    return out
```

lunedì 20 marzo 2023 16:12

Compression

$$PARSE(CB,WB) = \begin{cases} \langle 0, B[1] \rangle ; i \leftarrow i+1 & \text{if } d=0 \\ \langle d, |a| \rangle ; i \leftarrow i+|a| & \text{if } d>0 \end{cases}$$

- Parsing:

```
PARSING(S): // S = s1 ... sn
    W ← {}; B ← S; i ← 1;
    while (i ≤ n) {
        <d, x> ← PARSE(B, WB);
        if d = 0 {
            B[1] ← x; // <0, B[1]>
            i ← i+1; }
        else if d > 0 {
            |a| ← x; // <d, |a|
            i ← i+|a|; }
        W ← WB[i, n]; B ← B[i, n]
        OUTPUT(<d, |a|, c>);
    } return;

PARSE(WB, B) {
    d ← 0; |a| ← 0, c ← B[1];
    while (d < |B|)
        d++;
        if LCP(W[|W|-i, |W|]B, B) > |a|
            |a| ← LCP(W[|W|-i, |W|]B, B);
    if d = 0
        return <d, B[|a|+1]>; // return <0, B[1]>
    else if d > 0
        return <d, |a|>; // r
```

- Encoding: As for LZ77 encode $\{d_1 \ldots d_n\}$

Decompression

```
DECOMPRESS(S') // S' = <d1, x1>, ..., <dn, xn>  (xn char or int)
    i ← 1.
```

```
DECOMPRESS(S') // S' = <d1,x1>,....,<dn,xn>  (xn char or int)
    i ← 1;
    S ← {};
    while i < |S| {
        out ← DECODE(S, <di,xi>);
        S ← [S, out];
    }
    return out;

DECODE(S, <d,x>)
    if d = 0
        out[1] ← x; // x = char
    else if d > 0 {
        for i = 0,...,|α|-1 // out = zeros[1,|α|]
            out[i+1] ← S[|S|-d+i];
    }
    return out.
```

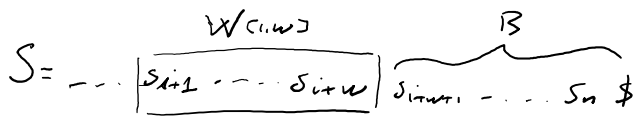
Gzip (fast LZ77)

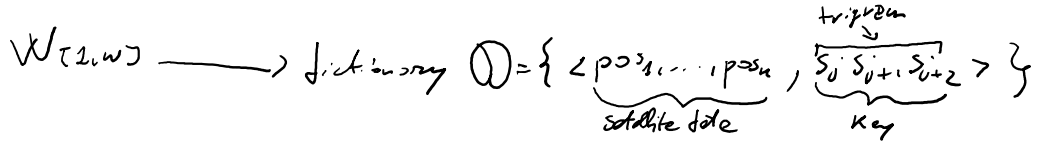
lunedì 20 marzo 2023 18:31

- Hash table (T) for speed up search on triplets

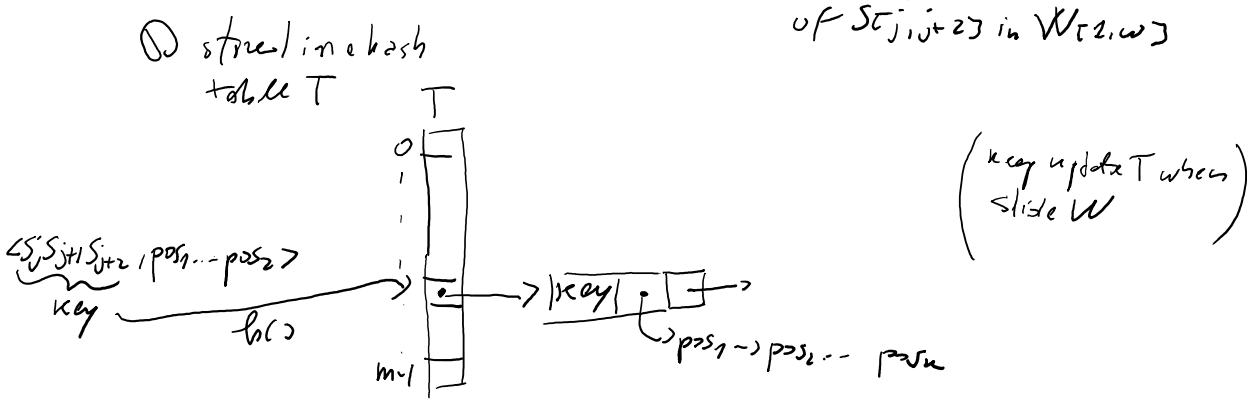
Compression

- Parsing:





pos_1, ..., pos_n = sorted positions of occurrences of S[j, j+2] in W[1,w]




```
PARSING(S): // S = s_1 ... s_n
    W ← {}; B ← S; i ← 1;
    T ← EMPTY(m); // empty hash table of size m
    while (i ≤ n) {
        α ← B[i, MAX(i+2, |B|)];
        <d, x> ← PARSE(α, T);
        if d = 0 {
            B[1] ← x; // <0, B[1]>
            l ← i+1; }
        else if d > 0 {
            |α| ← x; // <d, |α|>
            l ← i+|α|; }
        D_del ← TRIGRAMS(W[i, l]);
        D_add ← TRIGRAMS(B[i, l]);
        T ← DELETE(D_del, T);
        T ← INSERT(D_add, T);
        i ← l;
    } return T;
```

```
PARSE(α, T, WB)
    if (SEARCH(α, T))
        Pos ← SEARCH(α, T); // Pos = pos_1, ..., pos_k
    else
        return <0, B[1]>;
    for i = 1, ..., LEN(Pos)
        L[i] ← LCP(W[pos_i, |W|], B, B) // L[1,k] array of lengths of prefixes
    (i, |α|) ← MAX(L); // |α| = max{l}, i = argmax{l}
    d ← |W| - Pos_i + 1;
    return <d, |α|>
```

- Encoding:

  Building 2 Huffman

  1) H_1 for alphabet $\Sigma_1 = \{|\alpha_i|, c_j\}$ (lengths, literals)
     - $|\alpha_i|$ from $<d_i, |\alpha_i|>$
     - c_j from $<0, c_j>$

  2) H_2 for alphabet $\Sigma_2 = \{d_k\}$
     - d_k from $<d_k, |\alpha_k|>$

  Same i, k d_i distinguish $<0, c>$ with $<d, |\alpha|>$

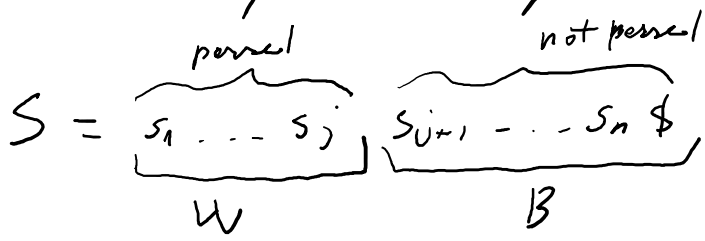
Decompression

```
DECOMPRESS(B)
    S ← ε;
    while (block ≠ EOF) {
        x ← DECODE(H_1, block)
        if x is literal // x = c_j (<0, c_j>)
            S ← [S, x];
        else // x is a length of prefix (<d_i, |α_i|>)
            d ← DECODE(H_2, block)
            S ← [S, DECOMPRESS(S, <d, x>)];
    }
    OUTPUT(S)
```

LZ78

martedì 21 marzo 2023 15:46

- Build incrementally a dictionary



We keep dictionary of W D = DICTIONARY(W)

D = {<id : f>}

↖ substring

Parsing

empty string

//Start : D = {<0 : ε>}, B = B[1,n] = S

PARSE(S, D)

while (B ≠ ϕ){

α, <id(α), c> ← PARSE(B, D);

OUTPUT(<id(α), c>); // id(α) id of LCP in D

D ← INSERT(<maxid+1 : αc>, D);

B ← B[|α|+1, n];

}

PARSE(B, D)

<id : f> ← LCP(B, D) // f LCP with B in D (f = α)

c ← B[|f|+1];

return f, <id, c>;

To manage dictionary keep $TRIE(\mathcal{D}) = (V, E)$

$u \in V \quad u = id(\alpha)$

$E \ni u \xrightarrow{c} u' \quad s.t. \quad \alpha' = \alpha c$

$id(\alpha) \quad id(\alpha')$

Decoding

Start: $T = \langle id_1, c_1 \rangle, \ldots, \langle id_n, c_n \rangle$, $\mathcal{D} = \{\langle 0 : \varepsilon \rangle\}$

DECODE$(T, \mathcal{D})$:

for $i = 1, \ldots, n$ {

$\langle id, x \rangle \leftarrow T[i]$;

$f \leftarrow$ SEARCH$(id, \mathcal{D})$; // $\langle id : f \rangle$ in $\mathcal{D}$

$\mathcal{D} \leftarrow$ INSERT$(\langle$ nextid $+1 : fx \rangle, \mathcal{D} \rangle)$;

OUTPUT(fx);

}

General

martedì 21 marzo 2023 16:53

- Not a compression algorithm "per se"
- It's a permutation of the input string s.t. the obtained string is more suitable to be compressed via simple algorithms
- The permutation forces locally homogeneous property to string
- Eventually Huffman or AC is performed to squeeze the bit-stream obtained from the simpler algorithms.

Forward transform

martedì 21 marzo 2023 17:12

Construct $s \longrightarrow bw(s)$

$S = s_1, \ldots, s_n$ $s_i \in \Sigma$ ordered alphabet, \$ s.t. $\$ < s \ \forall s \in \Sigma$

```
// start: M[0,n][0,n] matrix
FORWARDTRANSFORM(S)
    S ← S$; // append $
    for i = 0,...,n { // M[i][:] = string S cycled
        M[i][:] = s_{1+i} ... s_n $ s_1 ... s_{1+i-1};
    }
    M' ← SORTROW(M, Σ ∪ {$}); // sort lexicographically rows of M w.r.t. Σ ∪ {$}
    L ← M'[:][n]; // last column of M'
    (L̂, r) ← REMOVE($, L); // remove $ from L, r = pos of $ in L̂
    return (L̂, r); // bw(s)
```

Backward transform

martedì 21 marzo 2023 17:26

$S_\$ = S_1 \dots S_n \$$

Constant $bw(\cdot) \longrightarrow S$

$$M = \begin{bmatrix} S_1 \dots S_n \$ \\ S_2 \dots S_n \$ S_1 \\ \\ \\ \$ S_1 \dots S_n \end{bmatrix} \xrightarrow[\text{ROW}]{\text{SORT}} M' = \left[F \mid \dots \mid L \right]$$

Def $F := M'[:][0]$, $L := M'[:][n]$ $\quad (F = F[0,n), L = L[0,n))$

OSS 1 $\forall i = 1, \dots, n \quad M'[i][:] = Suff_n \$ S[i, k-1]$, $k = SA[i]$

$\quad = S[k, n]$

OSS 2 $L[i] = S[k] \Rightarrow F[i] = S[k+1] \quad \forall i = 0, \dots, n, \; i \neq 0 \; (L[0] = \$)$

($L[i]$ is the predecessor of $F[i]$ in S)

OSS 3 $\exists \pi_F, \pi_L$ permutations s.t. $F = \pi_F(S\$)$, $L = \pi_L(S\$)$ $\quad (\pi_F(i) = SA[i])$

OSS 4 $\sigma \in S \; (\sigma \in \Sigma')$ $\quad occ(\sigma, S) = \sigma_1, \dots, \sigma_n$ $\quad \sigma_i$ = i-th occurrence of σ in L

$pos(\sigma_i, L) < pos(\sigma_j, L) \iff pos(\sigma_i, F) < pos(\sigma_j, F)$

$(i < j)$

Def LF mapping $\quad LF : \{0, \dots, n\} \longrightarrow \{0, \dots, n\}$ permutation

$i \longrightarrow j$

if $L[i] = \sigma_k$ (k-th occurrence of σ in L) $\Rightarrow F[LF[i]] = F[j] = L[i] = \sigma_k$

```
// Start C = C[0, |Σ'|] = (0, ..., 0) indexed by Σ' ∪ {$} in lexicographic order
CONSTRUCT_LF(Σ', L)
    For i = 0, 1, ..., n-1
        C[L[i]]++; // C[σ] = #occ(σ)   (L[i] = σ)
    temp ← 0; sum ← 0;
    for i = 0, 1, ..., |Σ'| {  // C[i] = Σ_{j<i} C[j]
        temp ← C[i];
        C[i] ← sum;
        sum += temp;
    }
    for i = 0, 1, ..., n-1 {
```

- Extra space: $O(|\Sigma'|+1)$
- Time: $O(n)$

```
}
for i = 0,1,...,n-1 {
        LF[i] = C[L[i]];
        C[L[i]]++;  // increment for the new occurrence
}
```

Backward transform $bw(S) \longrightarrow S$

```
BACKWARD(bw, Σ)
    (L̂, r) ← bw;  // derive L for bw(S)
    L ← L̂[1, r-1] $ L̂[r+1, n];
    LF[0, n] ← CONSTRUCT_LF(Σ, L);
    k ← 0;
    i ← n-1;  // from n-1 to not output $
    while i ≥ 0 {  // L[LF[k]] predecessor of F[LF[k]] = L[k]
        S[i] ← L[k];  // (at start: S[n-1] = L[0])
        k ← LF[k];
        i--;
    }
```

- Space: $O(n)$
- Time: $O(n)$

Thm S can be reconstructed from its BWT in $O(n)$ time and space, eliciting possibly 1 cache-misses per symbol.

Move-To-Front transform (MTF)

martedì 21 marzo 2023 23:09

$S = s_1 \dots s_n$, $\Sigma = \{s_1, \dots, s_\sigma\}$ $s_i < s_{i+1}$

```
// Start L = (s_1, ..., s_σ) list of symbols, MTF = ""
MTF(S) // S = s_1 ... s_n
    for i = 1, ..., n {
        p ← SEARCH(S[i], L); // position of S[i] in L
        MTF ← [MTF, p]; s = L[p];
        if p > 0
            L[0] ← s;
        for i = 0, ..., p-1
            L[i+1] ← L[i];
    }
```

if $p > 0$ move to front $L[p]$ in L

- Working space: $O(|\Sigma|)$
- Time: $O(|S|) = O(n)$

} $(*)$

(*)

Thm MTF takes $O(|\Sigma|)$ working space and $O(n)$ time

Proof Working space: $O(|\Sigma|)$ ← for L

Time: $O(|S|) = O(n)$ ← for cycle ∎

Thm (no proof) $L_{MTF+\gamma} \leq 2H + 1$ and $L_{MTF+\gamma} \leq 2L_H$ (γ = γ-code, H = Huffman, $\mathcal{H}$ = entropy)

Thm $\exists S$ s.t. $|[MTF+\gamma](S)| < |H(S)|$ (of a factor $\Omega(\lg n)$)

Proof $S = 1^n 2^n \dots n^n$ $|S| = n^2$, $p(1) = \dots = p(n) = \frac{1}{n}$

For an optimal free-prefix code $\mathcal{C}$ $|\mathcal{C}(x)| = \lg_2 \frac{1}{p(x)} = \lg_2 n$

$|\mathcal{C}(S)| = |S| \cdot \lg_2 n = n^2 \lg_2 n$ bits $\approx |H(S)|$

($\mathcal{C}$: optimal)

MTF+γ: $S = 1^n 2^n \dots n^n \rightsquigarrow MTF(S) = 0^n 1 0^{n-1} 2 0^{n-1} 3 0^{n-1} \dots (n-1) 0^{n-1}$

$\rightsquigarrow MTF(S)+1 \rightsquigarrow \gamma(MTF(S)+1)$

$$|MTF+\gamma(S)| = \textcircled{4}O(n^2) + O(n\lg_2 n) = O(n^2 + n\lg_2 n)$$

$\textcircled{4}(n2) = \#0$

$\Rightarrow \textcircled{4}(n^2)$ times $\gamma(1)=1$ (1 bit)

$n = \#x, x>0, x<n$

$|\gamma(i)| = O(\lg i)$ bits $\Rightarrow \sum_{i \in n} |\gamma(i)| \leq O(\lg n)$

$$\Rightarrow |[MTF+\gamma](S)| = O(n^2 + n\lg_2 n) \overset{O(n^2)}{<} n^2\lg_2 n = |H(S)|$$

en dus $\dfrac{|H(S)|}{|[MTF+\gamma](S)|} = \dfrac{n^2\lg_2 n}{O(n^2)} = \Omega(\lg_2 n)$

Run-Length Encoding transform (RLE)

mercoledì 22 marzo 2023 00:24

$S[1,n] = \underbrace{s_1 \dots s_1}_{n_1}\ \underbrace{s_2 \dots s_2}_{n_2} \dots \underbrace{s_k \dots s_k}_{n_k}$ $\qquad \sum_{i=1}^{k} n_i = n$

$RLE(S) = \langle n_1, s_1\rangle, \dots, \langle n_k, s_k\rangle$

```
RLE(S)
    old ← S[1]; count ← 1;
    for i = 1,...,n { j ← i+1; new ← S[j];
            while (new = old){
                    count++;
                    j++;
                    new ← S[j];
            }
            i ← j;
            old ← new.
    }
```

Thm $\exists S$ s.t. $|[RLE+\gamma](S)| < |H(S)|$ by a factor $\Omega(\log n)$

Proof $S = 1^n 2^n \dots n^n$ $\quad |S| = n^2$ $\quad |H(S)| \cong |S| \cdot \mathcal{H} = n^2 \log_2 n$ ($p(x) = \frac{1}{x}$)

$RLE(S) = \langle n,1\rangle, \dots, \langle n,n\rangle$

$|\gamma(\langle n,i\rangle)| = O(\log n) \Rightarrow |[RLE+\gamma](S)| = O(n \log_2 n)$

$\Rightarrow \dfrac{|H(S)|}{|[RLE+\gamma](S)|} = \dfrac{\Theta(n^2 \log_2 n)}{O(n \log_2 n)} = \Omega(\log n)$

BZIP

mercoledì 22 marzo 2023 01:22

```
BZIP(S)
  (L̂, r) ← FORWARDTRANSFORM(S); // bw(S)
  S' ← MTF(L̂)
  for i = 1, ..., |S| { // S' = S'+1 for S'[i] ≠ 0
      if S'[i] ≠ 0
          S'[i] = S'[i]+1;
  }
  S' ← RLE0(S'); // RLE only on 0s of S'
  S' ← WHEELER(S'); // Wheeler code only on lengths of RLE0
  return S';

WHEELER(n)
    return B_{|Bin|-1}(n+1); // we code n+1 so that we can
                             // drop the first bit (n+1 ≥ 2)
```

Efficiency

- Avoid construction of M ($\mathcal{O}(n^2)$ working space) for BWT (Forward transform)

oss $L[i] = \begin{cases} S[SA[i]-1] & \text{if } SA[i] \neq 0 \\ \$ & \text{otherwise} \end{cases}$

Prof $F[i] = S[SA[i]]$ and $L[i]$ precedes $F[i]$ in S ◻

Thm S s.t. $|S| = n$. Exploiting oss the forward transform takes working space, time and I/Os needed for the SA construction (via qsort) (see SA_BUILD)

- working space: $O(n)$ ($O(n \log_2 n \text{ bits})$)
- time: $O(n^2 \log_2 n)$
- I/Os: $O(\frac{n}{B} n \log_M n)$

I/Os: $O(\frac{n}{b} \log_2 n)$