Algorithm Engineering

Notes for the course from University of Pisa

Aleandro Prudenzano

A.A. 2022-2023

Index

Chapter 1

Introduction

1.1 Introduction

1.1.1 Definition of algorithm

Following the *Knuth* definition: *an algorithm is a procedure which takes and input, produces an output and is built by a finite sequence of uniquely decodable and effective steps.*

- *procedure*, *input* $\rightarrow$ *output*: basically we are building a function: $f : I \rightarrow O$ which is also correct $\forall i \in I$ (this is the biggest difference with AI which have no correctness constraint);
- *uniquely decodable*: no ambigual sentences in algorithm description;
- *effective*: basic and atomic steps that can be executed in constant time (approximatively small time).

1.1.2 Method

We won't use any programming language to avoid syntactic sugar, instead we will use pseudocode and natural language (the less ambigous as possible).

We will build efficient and elegant algorithms to accomplish readability and bug-free means.

Our workflow will follow those steps:

- *problem abstraction*: we will go from problem description in natural language to some fancy formalism
- *design and analysis*: we will design and analyze a solving algorithm for that problem;
- *proof correctness*: we will prove that the proposed algorithm is in fact correct for the problem we are studying;
- *experiments*: we will check the correctness and check a posteriori if the prediction made in the analysis phase are in fact correct.

1.1.3 Von Neumann Model - RAM

In the past we have been exposed to the Von Neumann model of computation, also know as *Random Access Machine* in which we have read/write of single elements at a time from memory:

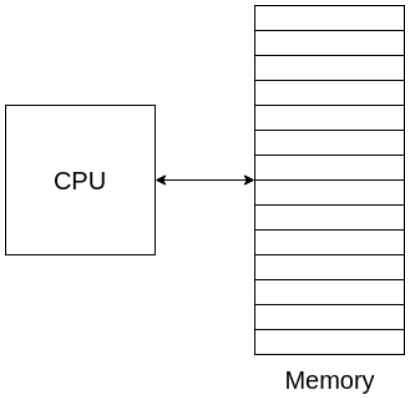


Figure 1.1: RAM model

In that model of computation the complexity is defined as:

$$T_A(n) = \#\text{steps for an input of size } n$$

This definition bring us some problems: we know nothing about the data except the size, if we have any informations about the data we can make better choice on which algorithm to use, moreover the time-complexity is used with asymptotic notation and for worst-case scenario which is not always the case.

1.1.4 2-Layer model

Unluckily Von Neumann architecture is not the case anymore because modern system are built on top of a *memory hierarchy*:

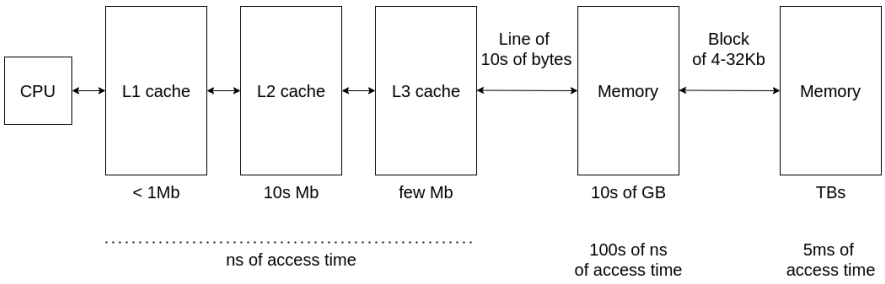


Figure 1.2: Memory hierarchy

Instead of the complex architecture above we will focus on a simpler abstraction:

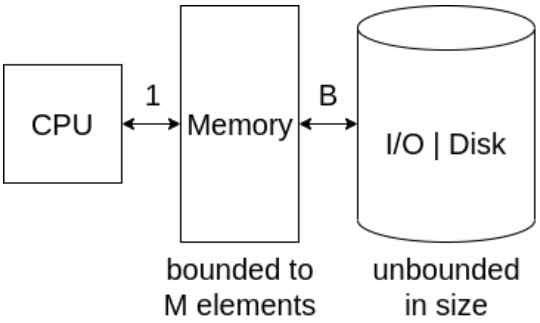


Figure 1.3: 2-layer model

In which an access in memory costs 0 and a costs in disk is called *I/O* and that's what we want to estimate.

In this model we try to enforce two properties:

- spatial locality: once we have loaded a block from I/O to memory we try to work on all the data on that block before moving on to another one;
- temporal locality: is a concept related to the working-set, in fact if it is small probably all my data will fit in memory and once the data has been loaded we won't need more I/O operation.

Enforcing these two properties we could diminuish I/O operations, so the time of the algorithm.

1.1.5 From time to size

Let's flip the notation: fixing the time T we want to know how much data we can process. Let's for example take 3 algorithms:

- $T_1(n) = n \implies n \leq T$
- $T_2(n) = n^2 \implies n \leq \sqrt{T}$
- $T_3(n) = 2^n \implies n \leq \log_2 T$

Of course the faster the algorithm, the more data we can process, but now we have a relation for data growth on fixed time. Knowing that relation we can argue on improvements if we improve CPU performance. Let's for example imagine we buy a k times faster machine, we will get:

- $n \leq kT$: improve by k
- $n \leq \sqrt{kT}$: improve by $\sqrt{k}$
- $n \leq \log_2 k + \log_2 T$: additive improve of $\log_2 k$ (additive improve means nothing!)

1.2 Some examples

1.2.1 Array lookup

Let's consider a sorted array of integers $A[1; n]$, we would like to to search for the element k, let's call this procedure $lookup(k)$.

Scan

The naivest algorithm is just the linear search. In RAM model it is $O(n)$ cause in the worst case we need to compare each element of the array.

In 2-layer model we need to evaluate the number of I/Os so if we have B elements in each block fetched from the disk we can: start from the left, bring a single block in internal memory and pay 1 I/O, then take the second block and pay 1 I/O and so on and so forth until we find the element inside one block.

In the end we pay $O(\frac{n}{B})$ I/Os in worst case.

Binary search

Since the array is sorted in RAM model we can use a binary search, which is $O(log_2 n)$.

In 2-layer model we can almost do the same: fetch the middle block and pay 1 I/O, let's assume we then go to the left, fetch another block paying 1 I/O and so on. For each jump we pay 1 I/O until we are left with a single block in which the following recursions are free from I/O fetches. So we do at most $O(log_2 n - log_2 B)$ fetches, which can be written as $O(log_2 \frac{n}{B})$ I/Os.

We can also see it as a binary search in the number of pages, which are of course $\frac{n}{B}$.

1.2.2 B+-Tree

We can further improve the access time using a $B^+ - Tree$: The deepest level contains all the values, the next level is formed taking the first element from each block, the next level is built in the same way but starting from the second deepest level, and so on.

The memory occupation is:

- n items for the deepest level
- $\frac{n}{B}$ for the second level
- $\frac{n}{B^2}$ for the next level
- ...
- $\frac{n}{B^i}$ for the i-th level

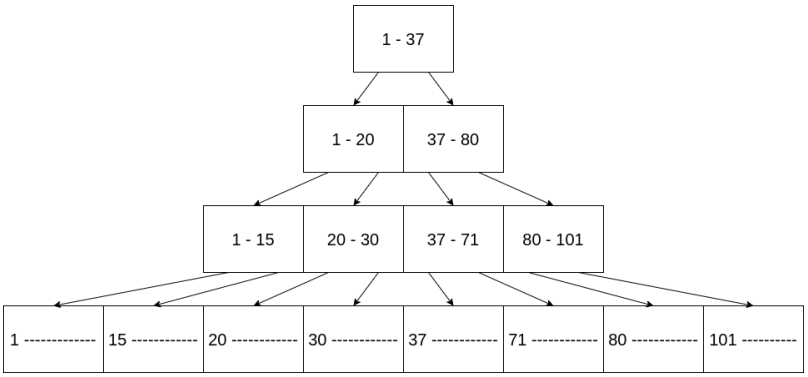


Figure 1.4: B+-Tree

We build levels until we reach a level in which we have a single block, that happens when:

$$\frac{n}{B^i} \leq B \implies n \leq B^{i+1}$$

so we can estimate the tree steep via:

$$log_B(n) \leq i+1 \implies i \geq log_B(n)+1$$

So we basically built a B-ary tree.

To search we start from the root, we bring that root block in memory and perform a search inside it to find the interval, then we fetch the next block and so on. We fetch a block for each level so $O(log_B n)$ I/Os. It's a huge improvement!

In RAM model

Let's consider the traversal in RAM model: for each of the $O(log_B n)$ I/O fetch we need to perform a binary search inside the block so in the end we have:

$$log_B n \cdot log_2 B = \frac{log_2 n}{log_2 B} \cdot log_2 B = log_2 n$$

Example

Let's consider those data: $n = 2^{30}$, $B = 32Kb$, 8 bytes of key and 8 bytes for the pointers:

- each page has $\frac{32kb}{16} = 2000$ values inside, which is the fan-out of the tree
- we pay $log_{2000} 2^{30} = \frac{log_2 2^{30}}{log_2 2000} \approx \frac{30}{10} = 3$
- with binary search we would have paid $log_2 \frac{2^{30}}{32Kb} = log_2 2^{15} = 15$

1.2.3 Sum of n integers

We have an array of integers $A[1; n]$ with n very large and A is stored on disk (we mean GBs or TBs of data) and we want to perform the sum of all the values inside A.

Linear Scan

The naivest approach is to use the scan algorithm , we iterate over the single blocks from left to right and we sum all the values inside the blocks. It's I/O complexity is $O(\frac{n}{B})$, it's an optimal algorithm because we can't do better since we need to access every item at least once.

Jump Scan

Let's think about another algorithm which jumps around to access items. Let's call it $Alg_{s,b}$ in which s is how much we jump each time and b is the block size of the algorithm, so the number of elements we process for each jump. So we have blocks of size B splitted in smaller blocks with size b (assume b divides B).

Our algorithm is:

- start from a block (of size b), sum all the items
- jump s blocks far away and start again until the end

Let's analyze the algorithm: the j-th block is $A[j \cdot b + 1; (j + 1) \cdot b]$ with $j = 0, 1, , \frac{n}{b} - 1$, let's use i to point to the block to process: $j = s \cdot i \mod \frac{n}{b}$ with $i = 0, 1, , \frac{n}{b} - 1$ (j is a permutation if and only if s and $\frac{n}{b}$ are coprime, so let's assume it). This definition of i and j says that we scans all blocks and we loop over them just once but:

- increasing s decreases locality because we process less elements at each jump
- increasing b increases locality because smaller blocks means smaller jumps

We have some edge cases:

- $s = 1$, b is whatever: we have the scan algorithm
- $s > 1$, $b = 1$: is the worst since we process a single element and then jump

Analyzing them from RAM model point of view those algorithms are all equivalent but in 2-layer models they are not. Let's assume $b \leq B$ and $s = 2$: we will exploit half of the blocks per page so we need a second pass to compute a full page!

In the end the number of passes is s if $s < \frac{B}{b}$ so the I/O complexity is $O(\frac{n}{B} \cdot min\{s, \frac{n}{B}\})$

1.3 Virtual Memory

Let's assume we have M items in memory and $(1+\epsilon)\cdot M$ elements on disk (M elements are both on disk and in memory). We want to calculate the probability that my algorithm will access the disk:

$$P(\epsilon) = \frac{\epsilon \cdot M}{(1+\epsilon)\cdot M} = \frac{\epsilon}{1+\epsilon}$$

That calculation is for a completely random algorithm which is not reasonable.

We can have a better estimation using some empiric data:

- we read/write not in constant time for $a \approx 30\%$
- we compute in constant time for $1-a \approx 70\%$
- assume C as the cost of an I/O operation

The average cost of a step is:

$$(1-a)O(1) + a[P(\epsilon)\cdot C + (1-P(\epsilon))\cdot O(1)] \approx a \cdot P(\epsilon)\cdot C$$

1.3.1 Example

Assuming:

- access in memory is $O(1) \approx 100ns = 10^{-7}s$
- access to disk is $5ms = 5\cdot 10^{-3}s$
- $C = \frac{5\cdot 10^{-3}}{100\cdot 10^{-9}} = 5\cdot 10^4$

the average cost of an operation is: $0.3 \cdot \frac{1}{1000}\cdot 5\cdot 10^4 = 15$.

The more the gap between memory and disk, the more the average cost, they are linearly dependant.

Chapter 2

Sorting

2.1 Sorting and permutation

The *sorting* problem gives us an array of values $A[1, n]$ and asks us to sort it with some comparative function.

The *permuting* problem gives us an array of values $A[1, n]$ and a permutation $\Pi[1, n]$ and asks us to implement Π over A: $A[\Pi[1]], A[\Pi[2]], A[\Pi[3]], A[\Pi[4]]$

Example: $A = [A, B, C, D], \Pi = [3, 1, 2, 4] \implies A_\Pi = [C, A, B, D]$

So in the first problem we are looking for the permutation and then we apply it, in the second we are just applying the permutation. In a paper from 1988 it is proved that in 2-layer model permuting and sorting are equivalent.

2.1.1 RAM Model

In RAM model we already know that sort problem has complexity of $O(nlogn)$ and permuting problem is $O(n)$.

2.1.2 2-level Model

Naive algorithm

The naivest algorithm would be:

```
for i = 1, n do
    B[i] = A[PI[i]]

A = B
```

so it is $\Theta(n)$ in I/O in the worst case since we can access the values of A in sparse order and each time we could hit a non resident block.

Reduce to sort

We can otherwise exploit the sorting algorithm to make less memory access:

```
Create couples X = <A[i], i>
Create couples Y = <PI[i], i> // source, destination

Sort Y for first element
Create couples Z = <X[1], Y[2]>

Sort Z for second element

permutation is Z[1]
```

The total cost is $3 \cdot O(\frac{n}{B} + 2 \cdot SORT(n)$ so $O(\frac{n}{B} + SORT(n))$

So permuting in 2-level model is actually $O(min\{n, \frac{n}{B} + SORT(n)\})$

Complexity of sort

Let's take sort complexity as it is because we will explain it in the future with some reasoning on the algorithms used:

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

Basically we say that we have $\frac{n}{B}$ as the cost of a single scan and $log_{\frac{M}{B}} \frac{n}{M}$ different scans.

Using the logarithm properties of change of base:

$$log_{\frac{M}{B}} \frac{n}{M} = \frac{log_2 \frac{n}{M}}{log_2 \frac{M}{B}}$$

So buying more memory the ratio decreases and so we need less runs!

Example: $M = 2^{30}$, $B = 2^{15}$, $n = 2^{60}$:

$$\frac{log_2 \frac{2^{60}}{2^{30}}}{log_2 \frac{2^{30}}{2^{15}}} = \frac{log_2 2^{30}}{log_2 2^{15}} = \frac{30}{15} = 2$$

This means that we sort Hexabytes of data in just 3 passes on a machine with 1GB of memory!

Let's call $L = log_{\frac{M}{B}} \frac{n}{M}$:

- permute is $O\left(min\{n, \frac{n}{B} \cdot L\}\right)$
- sort is $O\left(\frac{n}{B} \cdot L\right)$

so it is better to sort than permute when: $\frac{n}{B} \cdot L \leq n \implies B \geq L$

NB: B is $O(30Kb)$ meanwhile L is 2.

2.2 Multi-way Mergesort

2.2.1 Binary Mergesort

The already known binary mergesort sorting approach is based on splitting array in halves, when the partition has size equals to 1 the recursion stops and it starts the merging procedure which produces incrementally large sorted sequences. Just with that description we can notice that in the case of already sorted array we are doing useless work!

The pseudocode is:

```
Mergesort(A, i, j):
    if(i < j):
        m = floor((i+j)/2)
        Mergesort(A, i, m)
        Mergesort(A, m+1, j)
        Merge(A, i, j, m)
```

I/O complexity

Let's estimate I/O complexity of merge procedure (since the other part of the algorithm is just recursion): let's assume partitions of size l and a block size of B. We load a block from each partition, then those blocks stays in memory until we are done to compute the hole partition which gets replaced with the next one. So we do:

- $\frac{2l}{B}$ I/Os to read the data
- $\frac{2l}{B}$ I/O to write back the sorted data

so `Merge(A, i, j, m)` is $O\left(\frac{l}{B}\right)$

For each merge the cost is the cost of reading and writing two partitions of the size of that recursion level:

$$\frac{l_1 + l_2}{B} + \frac{l_3 + l_4}{B} + ... = \frac{l_1 + l_2 + l_3 + l_4 + ...}{B} = \frac{n}{B}$$

and we have $log_2 n$ recursive call since we split the array in half at each level, so the total I/O cost is:

$$O\left(\frac{n}{B} log_2 n\right)$$

This analysis is a bit weird since:

- M does not appear in the formula
- we are not taking into account the fact that in lower recursion level we are working always on the same block until the partition size doesn't get greater than the block size it self cause we don't need to fetch again:

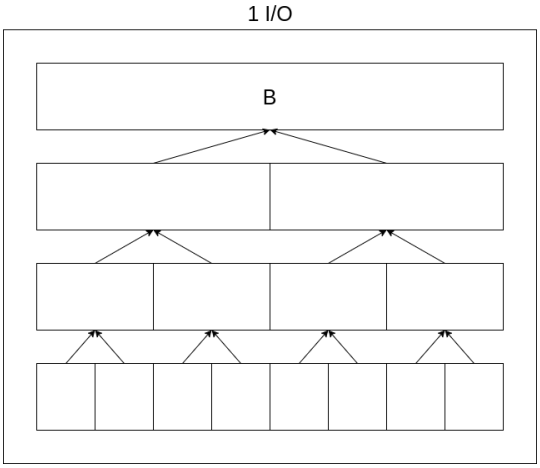


Figure 2.1: if partition size is less than block size, no new I/O is done

So we don't need to fetch blocks at each recursion. We have:

- $\frac{n}{B}$: cost for all the level with $l <= B$
- $\frac{n}{B}\left(log_2 n - log_2 B\right) = \frac{n}{B} log_2\left(\frac{n}{B}\right)$: cost for all the level with $l > B$

The previous argument can be extended to the whole memory instead of the single block: we have no I/O until we reach partition greater than the entire memory. So the total I/O cost is:

$$O\left(\frac{n}{B} log_2 \frac{n}{M}\right)$$

Avoid useless recursion

We can improve the algorithm to avoid some useless recursion: once we are in a partition which fits entirely inside the memory we can avoid other recursion call and insted sort the elements in place with another algorithm, for example heap-sort or quick-sort.

Avoiding useless recursion is nice to have since prevents memory usage avoiding the allocation of new stack frames.

2.2.2 Snow Plow

If we would be able to create longer sorted sequence we could do less recursion and so less scans and I/O operations.

Let's suppose to have $A = [1, 7, 5, 3, 8, 2, 4, 6]$ and $M = 3$, we create a set U for the unsorted items and a heap H for the sorted items. Let's fill the H with M items and keep U empty:

$$U = []$$

$$H = [1, 5, 7]$$

at each iteration we extract the minimum from the heap and insert another item from the array A following the rules:

- if the new item is lower than the last extracted item we insert it into U;
- if the new item is larger or equal than the last extracted item we insert it into H

Let's do a couple of runs:

- extract 1, insert 3 into H:

$$U = []$$

$$H = [3, 5, 7]$$

- extract 3, insert 8 into H:

$$U = []$$

$$H = [5, 7, 8]$$

- extract 5, insert 2 into U:

$$U = [2]$$
$$H = [7, 8]$$

- extract 7, insert 4 into U:

$$U = [2, 4]$$
$$H = [8]$$

- extract 8, insert 6 into U:

$$U = [2, 4, 6]$$
$$H = []$$

Once the heap is empty and the unsorted part is full we can terminate our run, so we transfer the elements from U to H and we start again to build the next sorted sequence.

Some pros of this algorithm are:

- this algorithm produces sequences of M elements or more;
- it's a kind of insertion sort, if the sequence is sorted it just runs in $O(n)$ time

Let's estimate on average the length of the produced sequences:

- we start from:

$$|U| = 0$$
$$|H| = M$$

- and we end with:

$$|U| = M$$
$$|H| = 0$$

Let's suppose that in a run we made τ reads, so we passed over $M + \tau$ elements, at the end of the run we have:

- τ elements written out;
- M elements remain inside U

To estimate the average we can suppose that on average half of the items will be inserted into U, and the other half will be inserted into H, so we will have $\frac{\tau}{2}$ elements going into U, so we can say that:

$$M = \frac{\tau}{2} \implies \tau = 2 \cdot M$$

So with the snow plow algorithm we are building sequence double the size of the ordinary binary merge-sort! In that way we are building less taller tree:

$$\#levels = log_2\left(\frac{n}{2M}\right) = log_2\left(\frac{n}{M}\right) - log_2 2 = log_2\left(\frac{n}{M}\right) - 1$$

Only one level less, but we have to recall that each level means a scan and with a lot of data scan means hours of computation!

Example: $A = [1, 8, 3, 2, 5, 0, 6], M = 2$

- fill U with M elements:

$$U = [1, 8]$$
$$H = []$$

- build the heap:

$$U = []$$
$$H = [1, 8]$$

- extract 1, insert 3 in H:

$$U = []$$
$$H = [3, 8]$$

- extract 3, insert 2 into U:

$$U = [2]$$
$$H = [8]$$

- extract 8, insert 5 into U:

$$U = [2, 5]$$
$$H = []$$

  we terminated the first run producing: $[1, 3, 8]$

- build the heap:

$$U = []$$
$$H = [2, 5]$$

- extract 2, insert 0 into U:

$$U = [0]$$
$$H = [5]$$

- extract 5, insert 6 into H:

$$U = [0]$$
$$H = [6]$$

- extract 6, nothing to insert:

$$U = [0]$$

$$H = []$$

  we terminated the second run producing: $[2, 5, 6]$

- build the heap:

$$U = []$$

$$H = [0]$$

- extract 0, nothing to insert. We ended the last run producing [0]

In the end we got 3 sorted sequences: $A = [1, 3, 8, 2, 5, 6, 0]$.

2.2.3 Multi-way mergesort

In terms of number of I/Os how many blocks we fetch for each partition to merge is the same since we are working only on the first blocks of the partitions because the other ones can be used only after the first are merged:

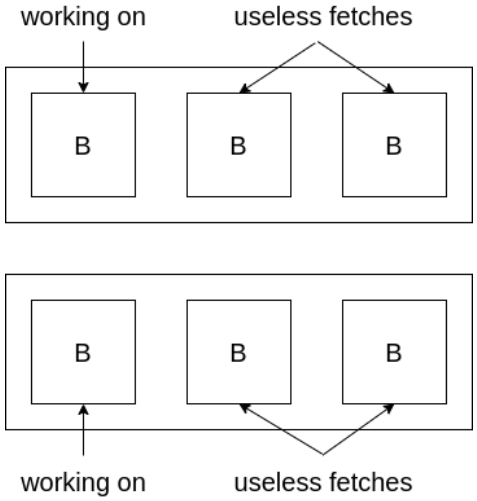


Figure 2.2: Useless fetches in binary merge-sort

We can exploit that feature to build a k-way merger:

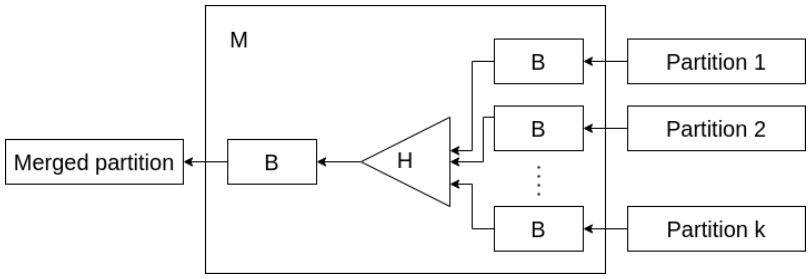


Figure 2.3: K-way merger

We need to get the minimum among k different values, so we use a min-heap to avoid comparisons, in that way extraction is $O(log_2 k)$ instead of $O(k)$

To refill the heap after one extraction we need to fetch from the same partition the extracted element came from.

Using this algorithm to merge k partition we just need a scan so $O(\frac{n}{B})$. Merging k sequences at a time is the same as having a recursion tree with k children for each node, so the tree height is $O(log_k \frac{n}{B})$ and so the number of scans to order the array. In the end the I/O complexity of the algorithm is $O\left(\frac{n}{B} log_k \frac{n}{B}\right)$.

Let's estimate k value: we can merge at most the number of blocks that fits inside the memory, remember that we need another one to build the output partition, so let's suppose that in memory we can have at most $k+1$ blocks:

$$(k+1)B \approx M \implies k+1 \approx \frac{M}{B} \implies k \approx \frac{M}{B} - 1 \implies k = O\left(\frac{M}{B}\right)$$

In the end the complexity of sort with k-way merge-sort is:

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

2.3 Sort lower bound

In RAM model we proved that there is a lower bound for sorting problem which is $\Omega(n \cdot logn)$, we proved it using binary decision tree:

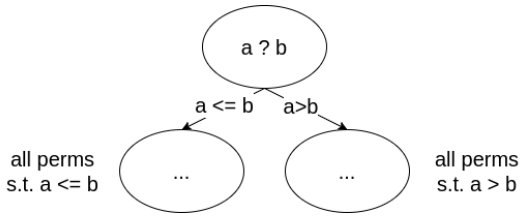


Figure 2.4: binary decision tree

and going down until we reach a node who contains a single partition. Of course the tree can be unbalanced but the narrowest tree is the balanced one, so the lower bound can be calculated using: $2^t \geq n! \implies t \approx O(n \cdot logn)$.

2.3.1 2-layer model

In this model each node can lead to several comparisons because each node is a fetch and in each block there are various elements that can be fitted in several different places.

Let's count how many ways we have to build the sequence: it's the same problem of asking in which positions will the single elements be placed in the sequence. That number is the dispositions: $\binom{M}{B}$, so that number is the number of comparison induced by 1 I/O operation. In this calculation though we have the bias that we know the right permutation of the elements, which is not true, so to count the effective ways we can sort the elements we must also consider the permutations of the single block, but only once, for the first time we see a new block.

We can say:

- fetch of new block induces $\binom{M}{B} \cdot B!$ comparisons, let's call it X;
- fetch of old block induces $\binom{M}{B}$ comparisons, let's call it Y

so we can build the new tree with both X and Y type nodes. We can have no more than $\frac{n}{B}$ nodes of type X since we can have at most that number of new loads. In an average path we have t loads so we have $t - \frac{n}{B}$ old I/Os. Assuming that the tree pushes all X type nodes above and all Y type nodes below we have:

- $\frac{n}{B}$ levels full of X type nodes, at each level we have X^i elements (with i the level depth)
- $t - \frac{n}{B}$ levels full of Y type nodes, with at each level $Y^j \cdot X^{\frac{n}{B}}$ elements (with j the level depth starting to count from the start of Y type nodes)

t of course must be large enough to produce $n!$ since we want to arrive at a leaf which contains only one possible array permutations:

$$X^{\frac{n}{B}} \cdot Y^{t-\frac{n}{B}} \geq n! \implies \left[\binom{M}{B} \cdot B!\right]^{\frac{n}{B}} \cdot \binom{M}{B}^{t-\frac{n}{B}} \geq n! \implies$$

$$\binom{M}{B}^t \cdot B!^{\frac{n}{B}} \geq n! \implies log_2\binom{M}{B}^t + log_2 B!^{\frac{n}{B}} \geq nlog_2 n \implies$$

$$tlog_2\binom{M}{B} + \frac{n}{B}Blog_2 B \geq nlogn \implies$$

let's use the asymptotic equality: $log_2\binom{a}{b} \approx b \cdot log_2 \frac{a}{b}$

$$tB \cdot log_2 \frac{M}{B} + \frac{n}{B} B \cdot log_2 B \geq n log_2 n \implies$$

$$tB \cdot log_2 \frac{M}{B} \geq n log_2 \frac{n}{B} \implies$$

$$t \geq \frac{n}{B} \frac{log_2 \frac{n}{B}}{log_2 \frac{M}{B}} \implies$$

$$t \geq \frac{n}{B} log_{\frac{M}{B}} \frac{n}{B}$$

so the lower bound for the sort problem is exactly the k-way merge-sort!

2.4 Example problem

Let's suppose we are a search engine and we have a bunch of url and we want to know if there are duplicate content inside those urls: this is the duplicate problem.

2.4.1 Hash + Trie

We can solve the problem hashing the content of the pages at the different urls and adding them to a trie and checking if there are duplicates. Scaling the algorithm to big data our complexity is: $O\left(\frac{n}{B}\right) + O\left(L \cdot n\right)$ in terms of I/Os operation, with L as the hash size. That's because the trie could become so large that we would need to do an I/O for each node in the tree traverse.

2.4.2 Hash + sort + scan

We can solve the problem hashing the content of each link, sorting the list and then scanning the list. That takes $O\left(\frac{n}{B}\right) + O\left(\frac{n}{B} log_{\frac{M}{B}} \frac{n}{B}\right)$.

2.4.3 General rule

If in the problem we have a lot of moving items (for example pointers) probably that problem can be solved in terms of sorting.

2.5 Multi-Disks

Let's assume we have 3 disks, so $D = 3$ and $K = 3$:

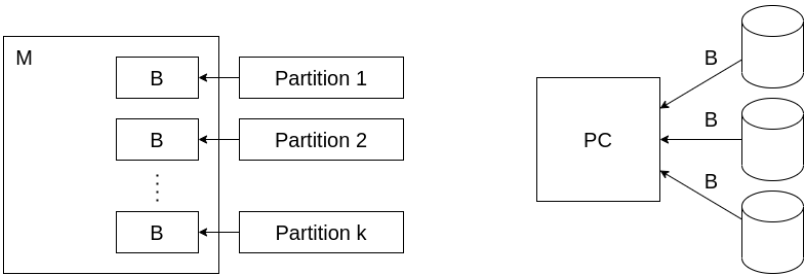


Figure 2.5: K-way merge-sort with multi-disks

If I load concurrently from the 3 disks in 1 I/O I fetch 3 blocks (pages). But that's true only if at each I/O we fetch from different disks! Merge sort in particular can be fully parallel but it's cursed (check notes for the full explanation).

Let's assume we have K runs and K disks, fully parallelism is not granted! It depends on the values in the pages. To obtain better performance we can create runs with round robin distribution in disks:

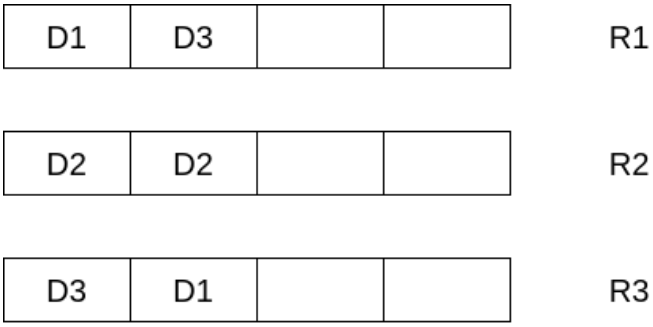


Figure 2.6: round robin distribution of runs in disks

What we can do to improve the efficiency without adding too much complexity is called *disk striping* and is already used inside Linux: we assume that all pages at the same time are part of a single large block of size $B \cdot D = B'$. Programs doesn't see multiple disks, it sees one big disks in which for each I/O we retrieve B' instead of B so our complexity becomes:

$$Sort(n, M, B') = O\left(\frac{n}{B'} log_{\frac{M}{B'}} \frac{n}{M}\right) = O\left(\frac{n}{DB} log_{\frac{M}{DB}} \frac{n}{M}\right)$$

To notice that the more disk i buy (increase D), the more runs we need to do, cause the log base is $\frac{M}{DB}$.

How much disk striping outperforms compared with the optimal algorithm,

which is $\Theta\left(\frac{n}{DB} log_{\frac{M}{B}} \frac{n}{M}\right)$?

$$\frac{\frac{n}{DB} log_{\frac{M}{DB}} \frac{n}{M}}{\frac{n}{DB} log_{\frac{M}{B}} \frac{n}{M}} = \frac{\frac{log_2 \frac{n}{M}}{log_2 \frac{M}{DB}}}{\frac{log_2 \frac{n}{M}}{log_2 \frac{M}{B}}} = \frac{log \frac{M}{B}}{log \frac{M}{DB}} = \frac{log \frac{M}{B}}{log \frac{M}{B} - log D} = \frac{1}{1 - log_{\frac{M}{B}} D}$$

since we fit DB inside the memory we can say $DB \leq M \implies D \leq \frac{M}{B}$ so we can say:

$$\frac{1}{1 - log_{\frac{M}{B}} D} \geq 1$$

so there is some outperforming but it's not that much until we get too many disks (eg. for $D = 1$ the algorithm is optimal), and moreover it is transparent to the algorithm!

2.6 Quicksort

The quicksort follows the distribution-based pattern. The pseudo-code is:

```
QS(S, i, j)
    if(i < j)
        m = Partition(S, i, j) // m is the position of pivot
        QS(S, i, m)
        QS(S, m+1, j)
```

the `Partition(S, a, b)` is a procedure which:

- chooses a "good" pivot and swaps it with the first element of the partition:

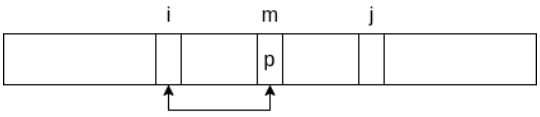


Figure 2.7: Swap pivot

- loops over the partition and builds it in the following way:

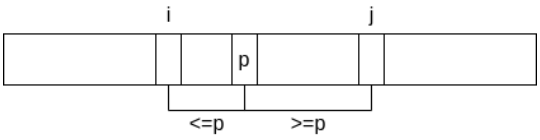


Figure 2.8: Build new partitions

We have different strategies to build our partitions:

- 1 pivot and 2 partitions;
- 1 pivot with 3 partitions:

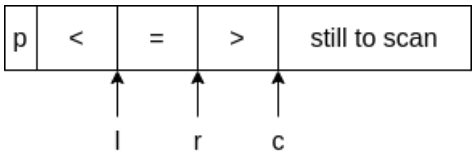


Figure 2.9: three-way partitioning

  in particular while scanning we can have:

  - $S[c] < p$: so we swap l with c and increment l;
  - $S[c] == p$: so we swap r with c and increment both r and c;
  - $S[c] > p$: so we just increment c.

  at the end we need to swap l with $pos(p)$ because p is mis-positioned.

- 3 pivot with 3 partitions: let's assume we have p', p'' and p''' and they are $p' < p'' < p'''$ so we use p'' as pivot.

Let's prove that in case of random choice of pivot we have optimal number of comparison:

$$X_{u,v} = \begin{cases} 1 & \text{iff } S[u] \text{ cmp } S[v] \\ 0 \end{cases}$$

we want the average number of comparison so:

$$\mathbb{E}\left[\sum_{u<v} X_{u,v}\right] = \sum_{u<v} \mathbb{E}\left[X_{u,v}\right] = \sum_{u<v} 1 \cdot \mathbb{P}\left(S[u] \text{ cmp } S[v]\right) + 0 \cdot \mathbb{P}\left(S[u] \text{ not cmp } S[v]\right)$$

$$= \sum_{u<v} \mathbb{P}\left(S[u] \text{ cmp } S[v]\right)$$

there is a comparison between $S[u]$ and $S[v]$ only if one of them is chosen as pivot but they can be compared later in recursion too, so the probability depends on the chosen pivot!

$$= \sum_{u<v} \mathbb{P}\left(S[u] \text{ is pivot or } S[v] \text{ is pivot}\right)$$

Let's take the sorted array to reason about the comparisons we can have:

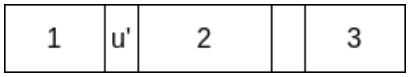


Figure 2.10: Array zones

- if pivot is in 1 or 3 there is the probability that u and v will be compared later in recursions;
- if pivot is in 2 then recursion will never compare u' and v'.

So we can have:

- $\hat{S}[u']$ is pivot:
$$\frac{1}{v' - u' + 1}$$
- $\hat{S}[v']$ is pivot:
$$\frac{1}{v' - u' + 1}$$
- $\hat{S} < p < \hat{S}[v']$: no comparison
- $p < \hat{S}[u']$ or $p > \hat{S}[v']$: recursion.

So
$$\mathbb{P}\left(S[u] \text{ cmp } S[v]\right) = \frac{2}{v' - u' + 1}$$
now we can rewrite:

$$\sum_{u<v} \mathbb{P}\left(S[u] \text{ is pivot or } S[v] \text{ is pivot}\right) = \sum_{u<v} \mathbb{P}\left(S[u] \text{ cmp } S[v]\right) = \sum_{u<v} \frac{2}{v' - u' + 1}$$

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

with the last inequality from the harmonic series.

2.6.1 Selection of the k-th ranked element

Assume we have $S = [2, 7, 6, 3, 8]$ and we want to get the 4^{th} ranked element, which is the k^{th} element in sorted array. We don't want to sort (which would take $O(nlogn)$) so we use the quicksort approach: we select a pivot m whose value is p and we do a single recursion to avoid the whole sorting cost. We can also exploit the three-way partitions:

- if k is in the low half part we recurse on it;
- if k is in the high half part we recurse on it;

NB: we know the position because p is in it's right place so we can compare m with k:

- $k < m$: search in lower part for k;
- $k > m$: search in higher part for $k - m$

The complexity of this algorithm is:

$$T(n) = O(n) + \mathbb{P}(\text{go to left}) \cdot T(n_<) + \mathbb{P}(\text{go to right}) \cdot T(n_>)$$

NB: in quick-sort formulation we have no probability since we recurse both on the left and on the right.

Let's restrict to good case: $n_<$ and $n_>$ both $\leq \frac{2}{3}n$, that means:

- $n_< \leq \frac{2}{3}n$: pivot is in the first $\frac{2}{3}$ of the array;
- $n_> \leq \frac{2}{3}n$: pivot is in the last $\frac{2}{3}$ of the array;

so we have pivot in $[\frac{1}{3}n, \frac{2}{3}n]$, and in the end the relation becomes:

$$T(n) \leq O(n) + \frac{2}{3}T\left(\frac{2}{3}n\right) + \frac{1}{3}T(n) \implies$$

$$\frac{2}{3}T(n) \leq O(n) + \frac{2}{3}T\left(\frac{2}{3}n\right) \implies$$

$$T(n) \leq \frac{3}{2}O(n) + T\left(\frac{2}{3}n\right) \implies$$

$$T(n) \leq O(n) + T\left(\frac{2}{3}n\right)$$

and according to Master's Theorem it's $O(n)$.

In terms of I/Os we have $O\left(\frac{n}{B}\right)$.

2.6.2 Bounded Quicksort

We have recursive calls which in the worst case are n, in that case we double the occupied space which is very bad. We can apply the technique known as *removal of tail recursion* rewriting the algorithm in this way:

```
BQS(S, i, j):
    while(j-i > n0){
        r = random pos in {i, _, j}
        swap S[r] and S[i]
        m = Partition(S, i, j)
        if (m <= (i+j)/2){
            BQS(S, i, m-1);
            i = m+1
        }
        else{
            BQS(S, m+1, j);
            j=m-1
        }
    }
    InsertionSort(S i, j) // when array is in cache
```

in this implementation the number of recursive call is always $O(logn)$:

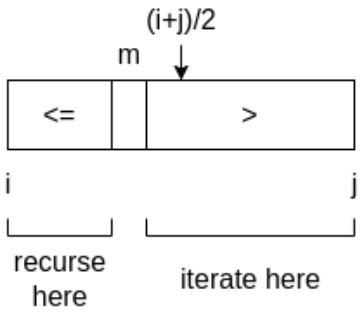


Figure 2.11: Bounded quick-sort

2.6.3 Multi-way Quicksort

We can build a new version of quicksort with the multi-way approach:

- pick $k-1$ different pivots: S_1, S_2, _, S_{k-1} supposing $S_0 = -\infty$ and $S_k = \infty$. It takes $O\left(\frac{n}{B}\right)$;
- sort them: $-\infty = S_0 < S_1 < \_ < S_{k-1} < S_k = \infty$. It takes $O(klogk)$;
- distribute the array S among the different S_is. We basically scan from left to right and build some blocks such that $B_i = \{S[j] : S_{i-1} < S[j] \leq S_i\}$:

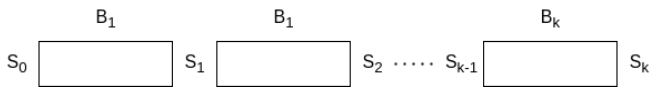


Figure 2.12: Multi-way partitioning

We need to allocate those blocks but how can we know the size to allocate? We can start to scan S (in $O\left(\frac{n}{B}\right)$), we find the block in which we need to insert the element (in $Ologk$). We allocate one block B for each of the B_i we are working with, when one block is filled we flush it in memory and fetch another one to fill:

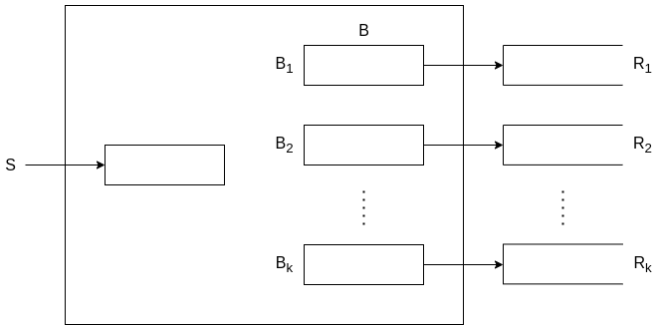


Figure 2.13: Multi-way partitioning in practice

And then we go recursively for each B_i: instead of binary way we go k-way. We can have at most $k = \Theta\left(\frac{M}{B}\right)$ so we have $O\left(log_k \frac{n}{B}\right) = O\left(log_{\frac{M}{B}} \frac{n}{B}\right)$ recursive calls and each of them is $O\left(\frac{n}{B}\right)$ since we do a complete scan of the array. That's true in the average case!

We would like to have balanced partitions with $\frac{n}{k}$ elements in each partition so we can't take k pivots randomly, it would not work for our case. We can do *oversampling*: we pick $(a+1) \cdot k - 1$ samples at random from S, we sort them, we pick as pivots $S_i = Sorted[(a+1) \cdot i]$ with $i = 1, 2, \_, k-1$. So we pick as pivots elements with at least $(k+1)$ distance from one another.

Let's prove that buckets are always good: let's estimate the probability that one of the bucket is bad:

$$\mathbb{P}\left(\exists B_i : |B_i| \geq \frac{4n}{k}\right)$$

considering S_{sorted}:

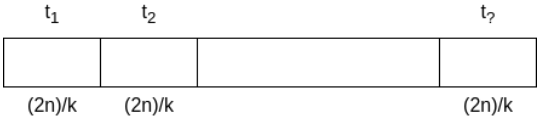


Figure 2.14: S_{sorted}

we have as much as $\frac{n}{\frac{2n}{k}} = \frac{k}{2}$ blocks. $|B_i| \geq \frac{4n}{k}$ means that a logical block (t_i) is totally included so: $S_{i-1} < B_i \leq S_i$:

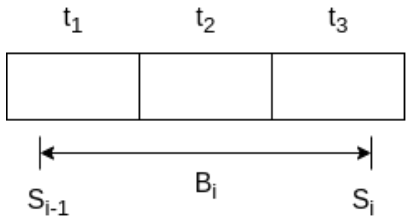


Figure 2.15: t_2 is fully included in B_i

but in B_i there are at least a elements, the sorted samples, so:

$$\mathbb{P}\left(\exists B_i : |B_i| \geq \frac{4n}{k}\right) \leq \mathbb{P}(\exists t_j : t_j \text{ contains } < a+1 \text{ samples })$$

$$\leq \frac{k}{2} \cdot \mathbb{P}(\text{specific } t_j : t_j \text{ contains } < a+1 \text{ samples })$$

Now we need to estimate the average number of samples inside a block:

- the probability for an item to appear inside a block (let's say t_2) is:

$$\mathbb{P}(\text{sample occurs in } t_2) = \frac{\frac{2n}{k}}{n} = \frac{2}{k}$$

- let's say $X = \#\text{samples in } t_2$ and:

$$X_i = \begin{cases} 1 & \text{if } S[i] \text{ falls in } t_2 \\ 0 & \text{otherwise} \end{cases}$$

  then $X = \sum_{i=1}^{n} X_i$. Now we can calculate the average number of samples inside a block:

$$\mathbb{E}[X] = \sum_{i=1}^{A} \mathbb{E}[X_i] = \sum_{i=1}^{A} 1 \cdot \frac{2}{k} = \frac{2}{k} A$$

  with $A = (a+1)k - 1$ as the number of samples.

Now we can estimate:

$$\mathbb{P}(t_2 \text{ contains } < a+1 \text{ samples }) = \frac{2}{k} A = \frac{2}{k}\left[(a+1)k - 1\right] = 2(a+1) - \frac{2}{k}$$

knowing that $k \geq 2$ we say that $\frac{2}{k} \leq 1$ so:

$$\mathbb{P}(t_2 \text{ contains } < a+1 \text{ samples }) \geq 2(a+1) - 1 \geq \frac{3}{2}(a+1)$$

Again:

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

$$\frac{2}{3}\mathbb{E}[X] \geq (a+1) \implies$$
$$\left(1 - \frac{1}{3}\right)\mathbb{E}[X] \geq (a+1)$$

So using the previous result we can upper bound:

$$\mathbb{P}(t_2 \text{ contains } < a+1 \text{ samples }) = \mathbb{P}\left(X < (a+1)\right) \leq \mathbb{P}\left(X < \left(1 - \frac{1}{3}\right)\mathbb{E}[X]\right)$$

What is the probability of having a value which is strictly less than a factor of the average? We can use the *Chernoff's bound*:

$$\mathbb{P}(X < (1-\delta)\mathbb{E}[X]) \leq e^{\frac{-\delta^2}{2}\mathbb{E}[X]}$$

applying that bound we reach:

$$\mathbb{P}(t_2 \text{ contains } < a+1 \text{ samples }) \leq e^{-\frac{1}{18}\mathbb{E}[X]} \leq e^{-\frac{1}{18}\cdot\frac{3}{2}(a+1)} = e^{-\frac{a+1}{12}}$$

In the end we have:

$$\frac{k}{2} \cdot \mathbb{P}(t_2 \text{ contains } < a+1 \text{ samples }) \leq \frac{k}{2} \cdot e^{-\frac{a+1}{12}} = \frac{k}{2}e^{-\frac{12ln(k)}{12}} = \frac{k}{2}e^{-ln(k)} = \frac{k}{2k}$$
$$= \frac{1}{2}$$

The probability of having a wrong partitioning is less than $\frac{1}{2}$ so we oversample, we build the buckets, if we are in a bad case we re-run the oversample and partition again.

Chapter 3

Random Sampling

3.1 Problem

Given n items: $S = [i_1, i_2, _, i_n]$ and a positive integer $m \leq n$, the goal is to select m items of S *uniformly at random* (each item can be picked with probability $\frac{1}{n}$).

There are two variants:

- n known;
- n unknown.

We will analyze the solution in 2 different computing models:

- 2-level memory model;
- streaming model: limited memory so you must decide for the element one at a time.

We assume `rand(a, b)` to return a random value in the range $[a, b]$ with uniformly distribution. To pay less I/Os we assume to access chosen elements from left to the right in sorted order for position.

3.2 Known n, 2-level memory

We have $S[1, n]$ with pointers to actual items, since we want to avoid to work on real S we copy it's content on S' and then execute the algorithm:

```
for s in {0, _ , m-1}
    p = rand(1, n-s)
    pick S'[p]
    swap S'[p] with S'[n-s]
```

Ex:

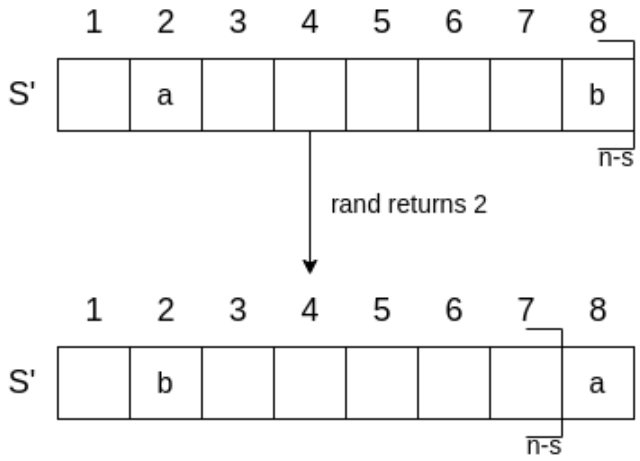


Figure 3.1: random sampling with $n = 8$

As we can notice at each step we exclude an element moving it at the end of the array. At the end of the algorithm the end of the array will contain the selected items.

This algorithm runs in linear time but jumps around in the array.

We would like to get an algorithm which provides only the indexes of the elements so that we can avoid access to disk:

```
D = {}
while |D| < m:
    p = rand(1, n)
    if p not in D:
        insert p in D
```

This algorithm has a huge problem: the more i go ahead, the more probable it is to find an element which is already in D:

$$P(p \in D) = \frac{|D|}{n} < \frac{m}{n}$$

in particular if $m \leq \frac{n}{2} \implies P(p \in D) < \frac{1}{2}$

NB: supposing $m < \frac{n}{2}$ is okay since if we have $m > \frac{n}{2}$ we could reverse the problem choosing the element that we won't take, so the upper bound to consider is always $m = \frac{n}{2}$.

So we have $O(m)$ expected time, and on average 2 runs for each p, in terms of I/Os: $min\{m, \frac{n}{B}\}$.

3.3 Known n, streaming model

I see $i_1, i_2, \_, i_n$ and I want to decide to keep or not for each element when I see it, without the capability to store them. Let's suppose $m = 1$ and we want to choose:

- $i_1 \to \frac{1}{n}$;
- $i_2 \to \frac{1}{n-1}$;
- $i_3 \to \frac{1}{n-2}$;
- ...
- $i_n \to \frac{1}{n-n+1}$;

There is no possibility of not choosing any element since the last element has probability of $\frac{1}{n-n+1} = 1$.

NB: The predicate *keep an element with probability p* can be evaluated with $rand(0, 1) \leq p$ using the rand function as a function which returns a real number in the range $[0, 1]$.

To prove that the probability are correct let's estimate $\mathbb{P}(\text{picking } i_j)$ given that I've not picked any other items. Let' prove by induction that every element is chosen with probability $\frac{1}{n}$:

$$\mathbb{P}(\text{picking } i_j) = \mathbb{P}(\text{pick } i_j) \cdot \mathbb{P}(\text{not pick } i_1, i_2, \_, i_{j-1}) = \frac{1}{n-j+1} \cdot \left(1 - \frac{j-1}{n}\right)$$

$$= \frac{1}{n-j+1} \cdot \left(\frac{n-j+1}{n}\right) = \frac{1}{n}$$

For the generic m we pick i_j with probability:

$$\frac{m-s}{n-j+1}$$

with s the number of already extracted items.

Let's suppose $s = 0$ with m items left to see:

$$\mathbb{P}(\text{picking } i_{n-m+1}) = \frac{m-s}{n-j+1} = \frac{m}{n-(n-m+1)+1} = \frac{m}{m} = 1$$

so the last m items will surely be taken.

NB: the formula is the generic case of choosing an element with probability:

$$\frac{\#\text{elements we need to pick}}{\#\text{available elements}}$$

NB: The main problem with all the approaches we followed is that we extract n random numbers which is too much, we won't cover algorithms that extract just m random numbers but there exists.

Example: $S = a, b, c, d, e$, $n = 5$, $m = 1$, $p = 0.5, 1, 0.1, 0.3, 0.7$

- $a : 0.5 \leq \frac{1}{5} = 0.2 \implies$ discard item;
- $b : 1 \leq \frac{1}{4} = 0.25 \implies$ discard item;
- $c : 0.1 \leq \frac{1}{3} = 0.33 \implies$ keep item.

Suppose $m = 2$:

- $c : 0.1 \leq \frac{2}{3} \implies$ keep item;
- $d : 0.3 \leq \frac{1}{2} \implies$ keep item.

3.4 Unknown n, streaming model

3.4.1 Reservoir Sampling

```
R[1,m] = S[1,m] //we pick the first m elements in S

for each next S[j], j >= m+1:
    h = rand(1, j)
    if (h <= m)
        R[h] = S[j]
return R
```

We can prove the correctness by induction: the base case with $n = m$ items should produce probability of bringing those items equals to 1 and the first step of the algorithm solves it. The inductive hypothesis for $n-1$ is that every item i_j for $j = 1, 2, \_, n-1$ is picked with $\mathbb{P} = \frac{m}{n-1}$. Let's prove that every item i_j for $j = 1, 2, \_, n$ has $\mathbb{P} = \frac{m}{n}$:

$$\mathbb{P}(i_n \text{ is picked}) = \frac{m}{n}$$

because I extract `rand(1, j)` with $j = n$ and pick the item if `rand(1, n) <= m` so:

$$\mathbb{P}(i_j \text{ is picked}, j < n) =$$

$\mathbb{P}(i_j$ is in R at step $n{-}1 \wedge (i_n$ is not picked $\vee\, i_n$ is picked but i_j is not kicked out)) =

$\mathbb{P}(i_j \in R$ at step $n{-}1) \cdot [\mathbb{P}(i_n$ is not picked$) + \mathbb{P}(i_n$ is picked $\wedge\, i_j$is not overwritten)] =

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

$$\frac{m}{n-1} \cdot \left[1 - \frac{1}{n}\right] = \frac{m}{n-1} \cdot \left[\frac{n-1}{n}\right] = \frac{m}{n}$$

Ex: $S = a, b, c, d, e, f$, $m = 2$

- $R = [a, b]$, $h = 3$, c not chosen;
- $R = [a, b]$, $h = 4$, d not chosen;

- $R = [a, b]$, $h = 2$, e chosen instead of b;
- $R = [a, e]$, $h = 3$, f not chosen.

In this algorithm we are generating too many random numbers.
NB: another way of computing:

$$\mathbb{P}(i_n \text{ is not picked}) + \mathbb{P}(i_n \text{ is picked} \wedge i_j \text{is not overwritten}) =$$

is:

$$\mathbb{P}(\text{not extracting j}) = 1 - \frac{1}{n}$$

3.4.2 Heap-based approach

The main concept is: given $S = a, b, c, d, e, f, \_$ for each element we extract a probability in the range $[0, 1]$ and we keep an heap for priority with size m, so at the end we return the heap content.
Ex: $H = \{(b, 0.2), (a, 0.1)\}$

- $(c, 0.05)$: discard;
- $(d, 0.3)$: insert: $H = \{(b, 0.2), (c, 0.1)\}$;
- $(e, 0.9)$: insert: $H = \{(e, 0.9), (b, 0.2)\}$;
- $(f, 0.2)$: discard.

the returned elements are: d, e.

Chapter 4

Randomized data structures

4.1 Treap

Treap stands for tree (as in binary search tree) + heap: every node is composed by a key and a priority (and of course the pointers to the children nodes, if any). The key is used to make binary search tree while the priority is used for min/max heap.

Ex:

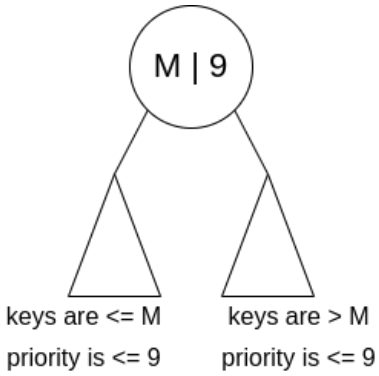


Figure 4.1: Key are letters, priority is an integer, max heap

It's a data structure highly used in computational geometry, for example let's assume we have a bunch of points:

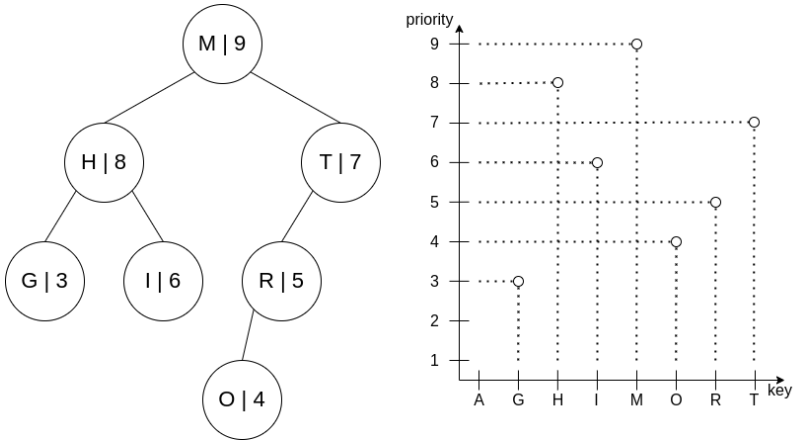


Figure 4.2: Treap as cartesian partitioner

each point splits the plane in two different parts and each part is a sub-tree of the treap. It's an efficient way to represent points for some kind of queries.

4.1.1 Search for a letter

It's a binary search tree for the letters so complexity is proportional to the height of the tree (NOT logarithmic): $O(h)$

4.1.2 Find max/min priority

Since the heap rules are observed we can do it in $O(h)$ too

4.1.3 3-sided range query

We want to find all the points in range $[a, b] \times [c, +\infty]$, so all points s.t.: $a \leq key \leq b$ and $c \leq priority$:

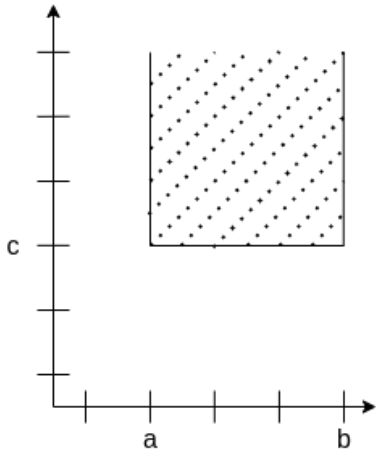


Figure 4.3: three-sided range query

being maximum heap for the priority when we go to a node we must check priority, if it's smaller than c we can stop recursion, then being a BST when we are on a node if the key is outside the range $[a, b]$ we can stop branching in both children. So we can execute a BFS and drop when the node doesn't match all requirements.

Ex: $[L, P] \times [5, +\infty]$

- visit M—9: matches criteria so returns node;
- visit H—8: key outside range so exclude left tree;
- visit I—6: key outside range so exclude all tree since no children;
- visit T—7: key outside range so exclude right tree;
- visit R—5: key outside range so exclude right tree;
- visit O—4: key inside range but priority outside so totally stop recursion.

Estimate cost

Let's try to estimate the complexity: in the worst case we reach nodes in the limits of the ranges so all the nodes outside the path are dropped (maybe at first there is the check). So in the worst case you check $O(h)$ nodes (two nodes for each node in the path) then for sure you visit all nodes inside the range until the priority is $> c$.

So the cost is:

- perculate the paths and drop lateral trees: $O(h)$;
- visit inside node until you don't exceed priority: $O(occurrences)$

in the end we have $O(h + occurrences)$. The first part is not so ok since the treap can be very tall but we can balance it using rotations, the second part is optimal since is proportional to the answer to the query.

4.1.4 Operations

Rotations

To insert and delete elements we need *rotations*:

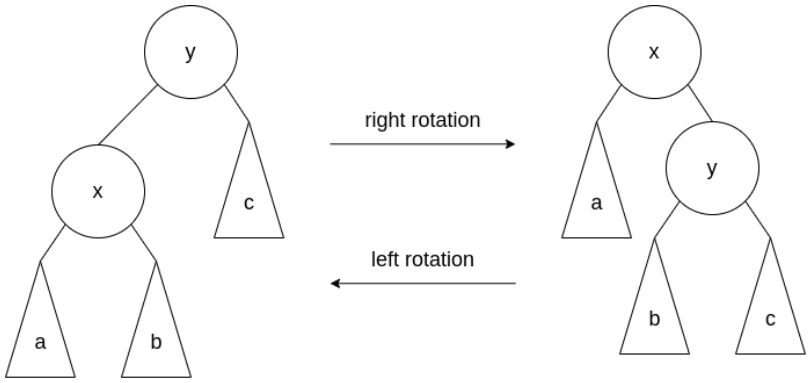


Figure 4.4: Rotations

We use right rotation when $priority(x) > priority(y)$ to promote x and we use left rotation where $priority(y) > priority(x)$ to promote y.

Insertion

To insert an element we need to:

- insert the new node respecting the key, as we would do in a BST;
- apply from the bottom to the top a series of rotation to comply with heap rules.

Ex: let's insert the node $(P, 6)$:

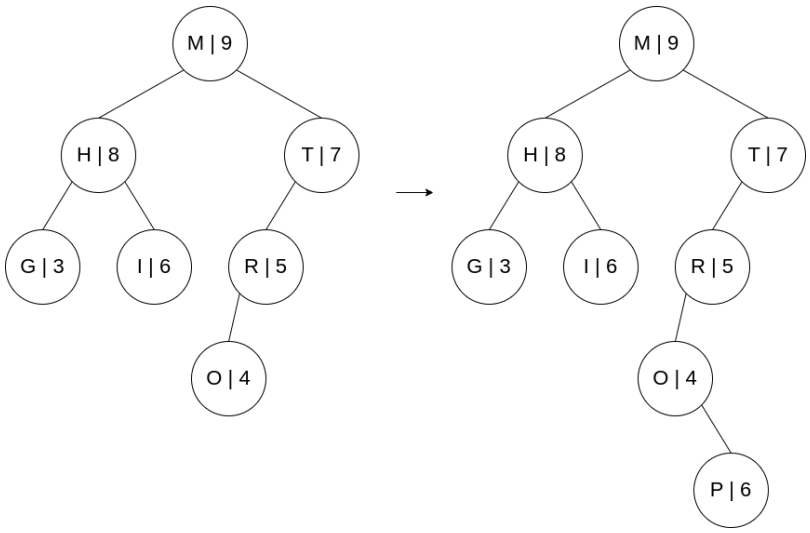


Figure 4.5: Insert the node respecting BST rules

we need to bubble up the node $(P, 6)$ since it's priority is bigger than $(O, 4)$:

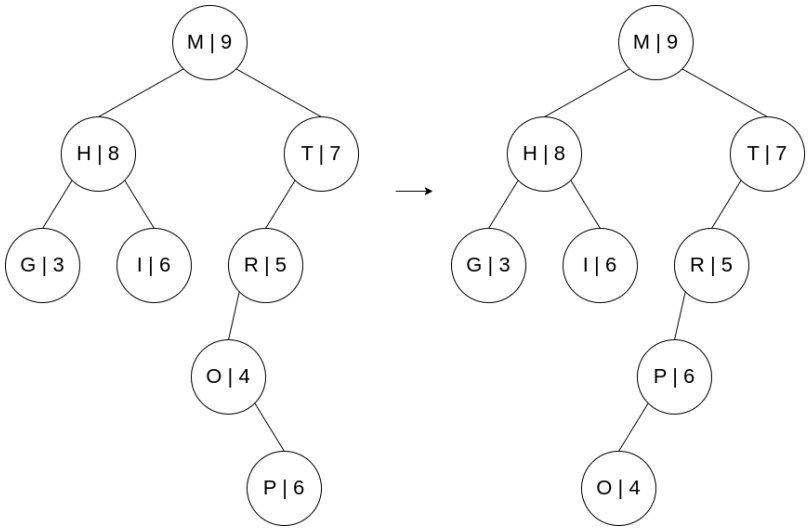


Figure 4.6: Left rotation

and now we need to bubble up that node again:

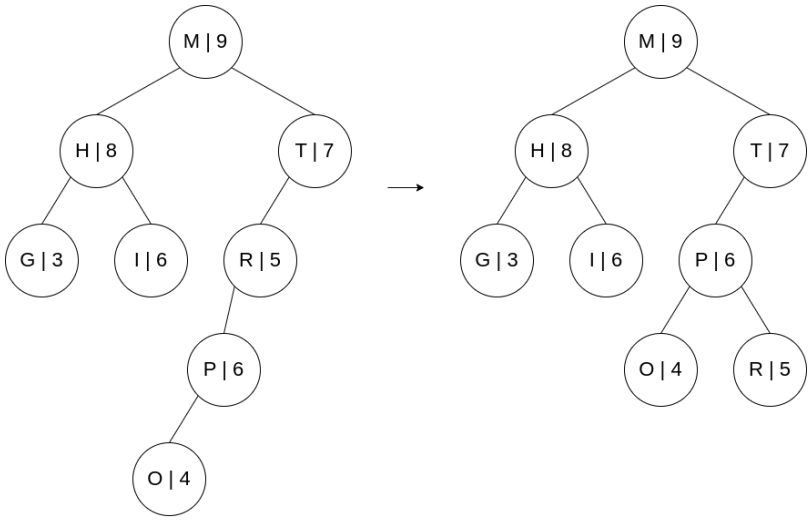


Figure 4.7: Right rotation

This operation is $O(h)$.

Deletion

To delete an element we exploit priority, as in heap:

- we search for the key;
- we change the priority of that node to $-\infty$ (if using a max-heap, otherwise we use $+\infty$);
- we push that element down using rotations;
- once the element is a leaf we can safely delete it.

This operation is $O(h)$.

Merge

The $merge(T1, T2)$ operation allows us to join two different treap in a single one if and only if all keys in $T1 <$ all the keys in $T2$:

- create a fake node with random key and priority $-\infty$ (if max-heap, otherwise $+\infty$);

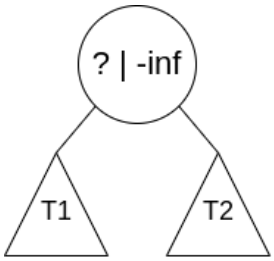


Figure 4.8: Fake node

- push the fake root down;
- once the fake node is a leaf we can safely delete it.

Split

The $split(k)$ operations allows us to split a treap in two different ones in which the first one has only the keys $\leq k$ and the second one has only keys $> k$:

- we insert a new node $(k, +\infty)$ (if max-heap, otherwise $-\infty$);
- we push the node up;
- once the node is the root we can delete the node and the two sons are the two new treaps;

4.1.5 Treap as BST

We can use the treap as a BST with random priority. This is useful to obtain a balanced BST on average without too much overhead, moreover it also supports split and merge!

Let's prove that if the priorities are chosen at random then $h = (logn)$ on average: take $x_1, x_2, \_, x_n$ as the keys sorted for priority, then we define:

$$A_k^i = \begin{cases} 1 & \text{if } x_i \text{ is a proper ancestor of } x_k \\ 0 & \text{otherwise} \end{cases}$$

then:

$$depth(X_k) = \sum_{i=1}^{n} A_k^i = \#\text{ancestors of } x_k$$

we want to know on average what's the value:

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

x_i can be ancestor of x_k only if $priority(x_k) > priority(x_i)$ so if $i < k$, we have 4 cases:

- x_i is root, it's a good case;
- $j < i$ and x_j is root so both x_i and x_k are in the right sub-tree, we can't decide, we need to recurse;
- $j > i$ and x_j is root so both x_i and x_k are in the left sub-tree, we can't decide, we need to recurse;
- $x_i \leq x_j \leq x_k$ and j is root, surely x_i is in the left sub-tree and x_j is in the right sub-tree, so surely x_i is not an ancestor of x_k.

We are left with a single case: $x_i, \_, x_j, \_, x_k$ whose probability is: $\frac{1}{|k-i|+1}$ in which $|k-i|$ is the size of the subset $\{x_i, x_{i+1}, \_, x_k\}$, so in the end:

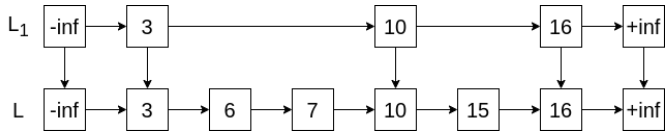
$$\sum_{i=1}^{n} \mathbb{P}(A_k^i = 1) = \sum_{i<k} \frac{1}{k-i+1} + \sum_{i>k} \frac{1}{i-k+1} \approx log_2 n$$

since it's an harmonic sum.

4.2 Skip list

Skip list is widely used inside dictionaries, for example is a core part of redis: a key-value storage (a dictionary which implements insert, delete and search for a key).

We start from a list and keep the good insertion and deletion policy optimizing the search. To improve search we create levels of list:



If we would partition every k elements each search would cost us:

$$k + \frac{n}{k}$$

in which the first term is referred to the search in the L_1 and the second term is referred to the search in the L_0 partition.

The optimal choose of k would be $k = \sqrt{n}$ but to keep that partition schema we should apply some algorithms during the insertion and deletion parts, we want to avoid to add that logic. To avoid those shenanigans we randomize the data structure: during the construction of the overlay we toss a fair coin for each element in the list:

- if we get tail: we copy the element from the list to the upper level;
- if we get head: we don't.

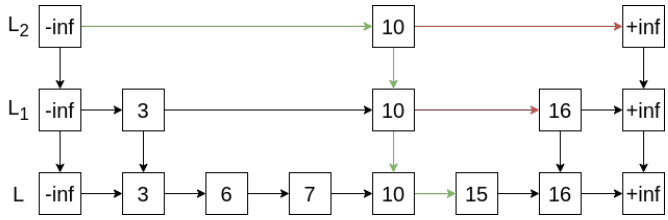
Using that algorithm we will build a newer level with $\approx \frac{n}{2}$ elements because the probability for each element of being chosen is $\frac{1}{2}$. Of course nothing prevents us to iterate the process again and again until we get a level with a constant number of elements!

We will prove that to have a constant number of elements on a list we need $O(log_2 n)$ levels on high probability.

4.2.1 Search

To search for a key we start from $-\infty$ at the highest level and search like in all the lists. If we can't find the element we are looking for we stop at the first item greater than the key and start to search in the next level but in the the new range we get with the last element smaller than the key and the first element greater than it. We go on with this approach until we find the key or until we reach the last list $+\infty$.

Example: searching for element 15:



- we start at L_2 and we check 10, we move on;
- we check for $+\infty$, since $10 < 15 < +\infty$ we go down from 10;
- we start at L_1 from 10 and we check 16, since $10 < 15 < 16$ we go down from 10;
- we start at L from 10 and we check 15, we found a match ad we stop here returning the node.

Basically: go forth if next node is less than the target, otherwise go down.

The cost for search operation is #horizontal pointers + #vertical pointers (the last one is the height of the structure, so the number of levels).

4.2.2 Deletion

We search for key, once we find it we delete the column associated with that element and connect the previous node with the next one, for each level in which the target element exists.

4.2.3 Insertion

First we find the position in which we should insert the element at the last level, then we insert the element in L and we toss a coin to choose if we have to add the element in the upper level too. NB: we insert the element after each last element checked for each level, we call that node *the frontier*.

4.2.4 Cost of search

Estimate the height

Let's prove that with high probability the height is $O(log_2 n)$: first we define $L = \#$levels of skip list on n items and we can say that:

$$L = max_k L(k)$$

with $L(k)$ the level of the k-th item. Then we can say that the probability for an item of having more than l levels is:

$$\mathbb{P}(L(k) > l) = \left(\frac{1}{2}\right)^l$$

which is the probability of having l consecutive tails. Moreover:

$$\mathbb{P}(L \geq l) = \mathbb{P}(max_k L(k) \geq l) = \mathbb{P}(\exists k : L(k) \geq l)$$

we can use the so called *union bound* to get an upper bound:

$$\mathbb{P}(\exists k : L(k) \geq l) \leq n \cdot \left(\frac{1}{2}\right)^l = \frac{n}{2^l}$$

With this relation we can distinguish two cases:

- $l \leq log_2 n \implies \frac{n}{2^l} \geq \frac{n}{2^{log_2 n}} = \frac{n}{n} = 1$: which is not so interesting as a case;
- $l > log_2 n \implies \frac{n}{2^l} < \frac{n}{2^{log_2 n}} = \frac{n}{n} = 1$: which is interesting since we want to know:

  $$\mathbb{P}(L \geq c \cdot log_2 n) \leq \frac{n}{2^{c \cdot log_2 n}} = \frac{n}{(2^{log_2 n})^c} = \frac{n}{n^c} = \frac{1}{n^{c-1}}$$

  and since $c \cdot log_2 n > log_2 n$ we are in this second case. That's the definition of *with high probability*, so the probability of having more than a $c \cdot log_2 n$ levels, and so failing, is very small.

Estimate the horizontal number of nodes in a search

Take the path we perculate during a search and traverse it backwards, from bottom to top, we can notice:

- starting from the bottom we can go to the left, so the node we are coming from in a normal search, until we find an upward pointer because once we find one we must go upward in the level.
- for each node the probability of having an upward pointer is $\frac{1}{2}$, since it depends on the result of a coin toss;

Since the average number of toss to get a tail is 2 we can say that on average we step 2 nodes horizontally for each level, so in total we have $2 \cdot log_2 n$ elements horizontally.

Total cost

So with high probability we have $O(log_2 n)$ vertical nodes and $O(log_2 n)$ horizontal nodes, in the end the search costs us $O(log_2 n)$.

NB: using this randomized data structure we achieve $O(log_2 n)$ search time avoiding the rotations we would have on a binary tree!

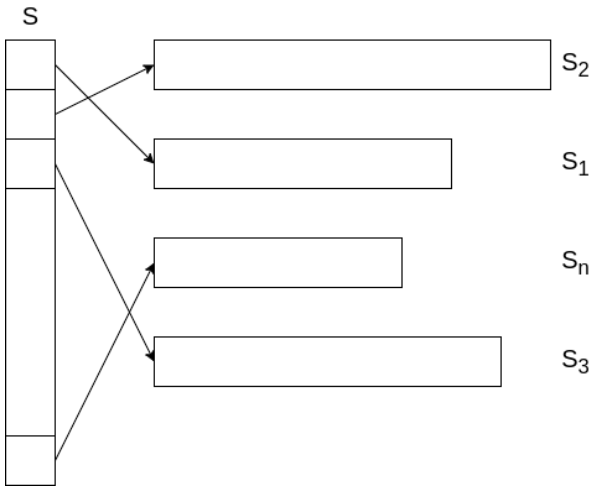
NB: skip list supports range queries too, so it's preferred to a dictionary in some contexts.

Chapter 5

String sorting

A string is a sequence of symbols/characters/letters (all equivalent), symbols are drawn from an alphabet Σ and we state that $\sigma = |\Sigma|$. Moreover in the sorting problem we have: $S[1, n]$ as our sequence of strings , N as the total length of all the strings, n as the number of strings and $L = \frac{N}{n}$ as the average length of the strings.

Strings are stored in memory in this way:



we could use the already analyzed `qsort` procedure:

```
qsort(array, #items, size_of_obj, cmp_function)
```

in which we sort the pointers which points to the string and the comparison function analyzes the strings.

Qsort is a comparison based algorithm which executes $O(nlog_2 n)$ comparisons, since we are dealing with strings each comparison is $O(L)$ so the total cost of sorting using qsort is $O(L \cdot n \cdot log_2 n) = O(N \cdot log_2 n)$.

In I/O model tough we have some drawbacks: since in quick-sort when $[l, r]$ is small enough we apply insertion sort to exploit cache, we lose every boost because strings are stored away from the pointer array, so no improvement.

5.1 Lower bound

We can prove a lower bound for string sorting:

- we must sort the first character of each string to begin with, so $\Omega(n \cdot log_2 n)$;
- let's suppose to have:
  - S_1 =abaco;
  - S_2 =abate;

– S_3 =abb.

we can estimate the number of characters we need to scan to lexicographically sort those strings:

– for S_1: $d_1 = 4$;
– for S_2: $d_2 = 4$;
– for S_3: $d_3 = 3$.

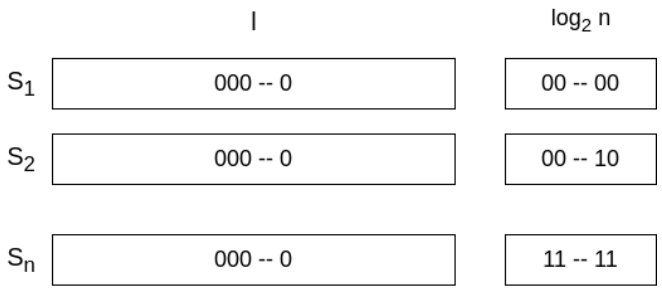
we call those numbers *distinguishing prefix*, moreover we define $d = \sum_s d_s$ as the totality of those prefixes. In the end for this part we have $\Omega(d)$.

So a theoretical lower bound for this problem is:

$$\Omega(n \cdot log_2 n + d)$$

in terms of time/comparison.

It's a bit like having strings like:



for each of those strings we have $d_i = l + log_2 n$, so $d = \sum_s (l + log_2 n) = l \cdot n + n \cdot log_2 n$, so qsort would take $O(L \cdot n \cdot log_2 n)$ since L is the average in our case we have:

$$O((l + log_2 n) \cdot n \cdot log_2 n) = O(l \cdot n \cdot log_2 n + n \cdot log_2^2 n)$$

, we have the term $log_2^2 n$ more than the lower bound, so qsort is at least a log_n factor from the optimal.

5.2 Radix Sort

We have two different approaches:

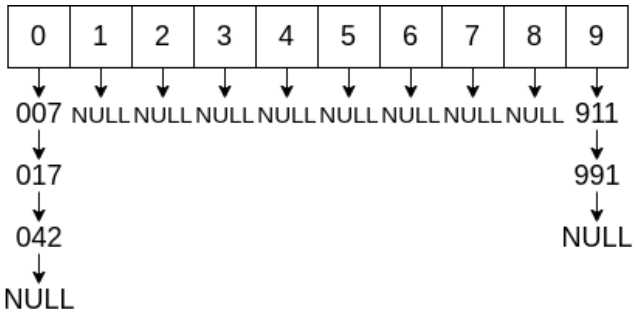
- Most Significant Digit first (MSD): in which we scan strings left to right;
- Least Significant Digit first (LSD): in which we scan strings right to left

5.2.1 MSD

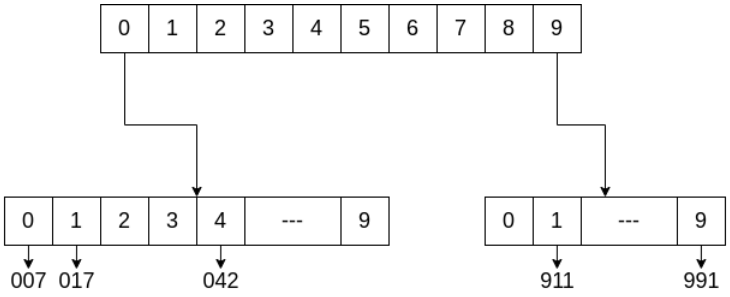
Let's start with an example:

$$S = \{007, 017, 042, 911, 991\}$$

we have $\sigma = 10$, $\Sigma = \{0, 1, \_, 9\}$, $n = 5$, $N = 15$. We start with an array of size σ, we scan each item and place it inside the right bucket according to the first character of each element:



then for each bucket we create a new array of size σ and repeat the process for each element in that bucket:



now we have just one string in each bucket, so scanning the leaves of this tree from left to right we have our strings sorted. Of course we don't finish at the same level for each bucket but we always build a tree-like structure.

An upper bound for the arrays we need to create can be: for each string we have we need to create at most d_s nodes, of course a lot of them are common between strings but we can state that:

$$\#nodes \leq d = \sum_s d_s$$

So total space complexity is $O(d \cdot \sigma)$, and since the building of the tree is the sorting itself the sorting time is the same as the space complexity, so $O(d \cdot \sigma)$.

Since the alphabet is constant we drop it from the big O notation, so in the end our time complexity for sorting is $O(d)$ which is lower than the upper bound we proved before. That's because the Radix sort is not a comparison-based algorithm and the lower bound we stated before is only applicable to those kind of algorithms!

Trie data structure

The tree we have built during radix sort is a particular data structure called *trie*. It is a tree in which each node has σ pointers to nodes, basically it's a σ-ary tree.

It's a data structure a lot efficient for searching operations but it's inefficient in space, which is $O(d \cdot \sigma)$, and if the alphabet is huge we have a lot of space, most of the time wasted for null pointers.

It is efficient in search operation because we traverse the tree, with the known key in $O(p)$ with $p = \#$length of the string we are looking for.

To avoid wasting space in null pointers in the case of big alphabets we can use an hash-table or a BST to store the tuples: $< character, ptr >$.

Using the hash-table approach we need to add an hash-table on each node whose size is proportional to number of pairs stored in that node (which is the number of edges outgoing from that node). Using an hash-table we could potentially have collisions but since size is proportional to keys access is still in $O(1)$ on average.

Using the BST we have access in $O(log_2\sigma)$ because we have at most σ nodes in BST, if it is balanced, of course.

Of course using one of the methods above lead us to lose the order between outgoing edges so if we want to use MSD Radix sort the algorithm becomes:

- build the trie in: $\sum_s O(1) \cdot d_s = O(d)$ on average;
- sort each hash table and traverse the edges in order, using qsort we'll have $O(\#edges \cdot log_2 \#edges)$ let's call e_u the average number of edges for each node: $O(e_u \cdot log_2 e_u)$.

so in the end:

$$\sum_u O(e_u \cdot log_2 e_u) = O\left(\sum_u e_u \cdot log_2 e_u\right)$$

we can upper bound $e_u \leq \sigma \forall u$:

$$O\left(\sum_u e_u \cdot log_2 e_u\right) \leq O\left(\sum_u e_u \cdot log_2 \sigma\right) = O\left(log_2\sigma \cdot \sum_u e_u\right)$$

and we have the sum of all the edges which is equal to the number of nodes, so we can upper bound it too:

$$O\left(log_2\sigma \cdot \sum_u e_u\right) \leq O(d \cdot log_2\sigma)$$

So this algorithm takes $O(d \cdot log_2\sigma)$ time and $O(d)$ space.

5.2.2 LSD

Given the sequence $S = \{017, 042, 665, 111, 007\}$ we scan the sequences backwards, exploiting a stable sort to sort the sequences by the first digit. Since we use a stable sort the elements with the same digit will keep the relative order among them. As stable sort we use counting sort which runs in $O(\#keys + |\Sigma|)$.

starting	$i = 3$	$i = 2$	$i = 1$
017	111	007	007
042	042	111	017
665	665	017	042
111	017	042	111
007	007	665	665

Table 5.1: LSD sorting in action

as we can see in the example above the strings are sorted at the end of the last step.

Complexity

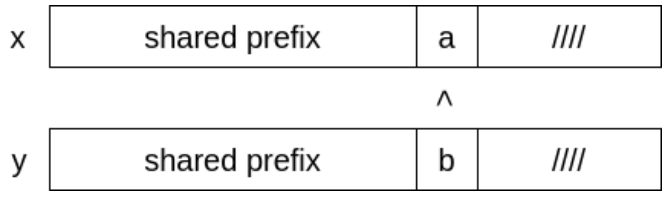
The total complexity is $\#digits \cdot O(\text{stable sort})$ so:

$$O(L \cdot (n + \sigma)) = O(n \cdot L + \sigma \cdot L) = O(N + \sigma \cdot L) = O(N)$$

so we have $O(N)$ time and $O(N)$ space.

Proof of correctness

A simple proof of correctness can be: let's start from two strings x and y, and we pose the hypothesis that $x < y$, we want to prove that if $x < y$ then the LSD radix sort will sort them according to that statement. We build those strings as generic:



since we scan from right to left we don't care of the content of the tail of those strings, once we get to a and b we sort them with x before y, then for all the characters in the shared prefix we use a stable sort so the relative order between x and y remains the same until the end.

Since it's true for each pair of strings it is true for all sets of strings.

A better complexity proof

Let's assume we have binary strings, so strings composed just with ones and zeroes: instead of comparing them bit by bit we group them by r bits, so we have $\frac{b}{r}$ groups each of r bits. Now we have $\sigma = 2^r$.

Applying counting sort over there will give us $\frac{b}{r}$ phases, each costing $n + 2^r$ so $O\left(\frac{b}{r}(n + 2^r)\right)$ which has minimum for $r = log_2 n$:

$$O\left(\frac{b}{log_2 n} \cdot (n + 2^{log_2 n})\right) = O\left(\frac{b}{log_2 n}(n + n)\right) = O\left(\frac{n \cdot b}{log_2 n}\right) = O\left(\frac{N}{log_2 n}\right)$$

So the actual complexity is $O\left(\frac{N}{log_2 n}\right)$.

So the original lower bound was $\Omega(n \cdot log_2 n + d)$ but we have a better lower bound of $O\left(\frac{N}{log_2 n}\right)$ because we aren't using comparison based approach.

NB: this algorithm is bad in case of short distinguish prefix, in that case MSD is better because those prefix are found faster!

5.2.3 Multi-key Quicksort

The algorithm is:

```
MKQ(R, i)
    if (|R| <= 1)
        return R
    else
        choose a pivot string p in R
        R_less = {s in R : s[i] < p[i]}
        R_equals = {s in R : s[i] = p[i]}
        R_greater = {s in R : s[i] > p[i]}

        A = MKQ(R_less, i)
        B = MKQ(R_equals, i+1)
        C = MKQ(R_greater, i)

        return A | B | C
```

We have an invariant for this algorithm: all the strings in the set R share $i - 1$ characters, so in terms of $i - 1$ characters, they are the same string. Then we concentrate on i-th characters: we put the strings in $R_<$, $R_=$, $R_>$ according to the lexicographical order of that character, in the end we partitioned those strings by that character that could be the last of the distinguish prefix. That's a verbose correctness proof.

Of course we invoke the procedure as `MKQ(S, 1)`.

Complexity

For sure we can say that:

- $|R_<|, |R_>| < |R|$: because at least p won't be inside one of them so the size of those sets will shrink;
- $|R_=| \leq |R|$: they are equals when all characters are the same one. That case is bad with recursion but on the next iteration we will go to the next index and since we recurse until $i \leq max_i|s_i|$, so it will terminate eventually.

Let's define $n = \#$strings and $N =$total length of all strings. As we said when picking $s \in R$ we have two cases:

- $s[i] < p[i]$ and so $s \in R_<$ and $s[i] > p[i]$ and so $s \in R_>$;
- $s[i] = p[i]$ and $s \in R_=$ but we increment i so after some recursion level we will fall in the first case.

first case is the same as quick-sort so we could even end up in linear number of recursions, in the worst case but for our analysis we will state that on average we will pick a random pivot such that we'll have:

$$|R_<| \approx |R_>| \approx \frac{|R|}{2}$$

so we will have $O(log_2|S|)$ recursive steps on average.

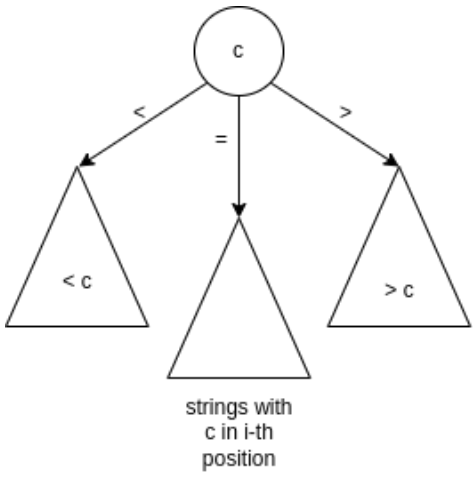
For the string s the number of recursive call is $O(log_2 n + d_s)$, in total we have:

$$\sum_s O(log_2 n + d_s) = O\left(\sum_s log_2 n + d_s\right) = O(n \cdot log_2 n + d)$$

so on average we meet the lower bound.

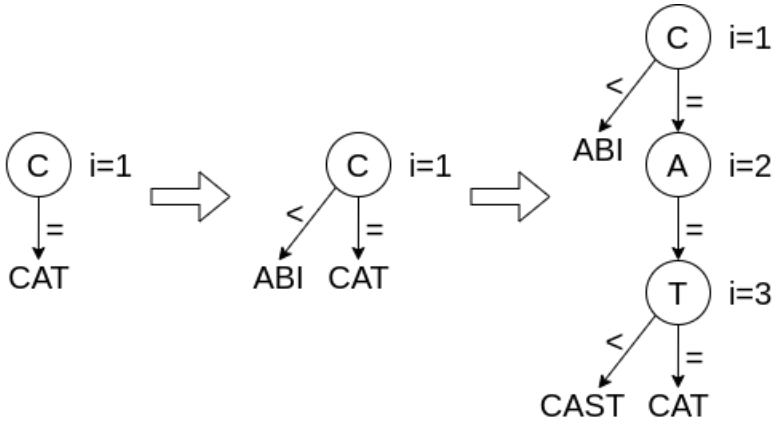
Ternary search tree

Starting from Multi-key quick-sort we can build a ternary search tree (or ternary trie) which has the following structure:



Each node has 3 pointers, going to the left and right arms we don't advance the position, we do it going in the middle element.

Example: insert the elements from the set $S = \{cat, abi, cast\}$:



in the third insertion we insert first the node A because *cat* and *cast* share the prefix *ca*, then we choose one of those strings (in this instance *cat*) and we store it in the third node, then we need to store *cast*, since $s < t$ we put that string on the left arm.

Of course TST could be unbalanced but if strings are inserted randomly we have that a string s gets a path of length: $O(|d_s| + log_2 n)$. Then each node needs only an array of 3 pointers and one splitting character.

Chapter 6

Set intersection

Given two sets $A, B \subseteq \mathbb{N}$ we want to compute:

$$A \cap B = \{x : x \in A, x \in B\}$$

That problem is useful in database JOINS operations and in search engines too where we have a dictionary for each search term and associated to it we have the list of document IDs who contains that search term, so when we are looking for results of a query we want the documents who contains all the terms, so the intersection between the dictionary associated elements.

Throughout all the algorithms we will see we assume that sets are sorted, because otherwise the only algorithm is the for loop inside a for loop which in case $|A| = n$ and $|B| = m$ it's $O(m \cdot n)$ time which is very bad!

6.1 Common approaches

6.1.1 Merge-based intersection

It's derived from the merge procedure from merge-sort. Take for example $A = \{1, 5, 8, 9, \_\}$ and $B = \{2, 5, 9, \_\}$, we take two pointers at the start of the lists and we move the one that points to the smallest element at each comparison, when both points to the same element we get a match and save it. So for example:

- we check 1 from A and 2 from B, no match, we move p_A;
- we check 5 from A and 2 from B, no match, we move p_B;
- we check 5 from A and 5 from B, we have a match, yield the item and move both p_A and p_B;
- we check 8 from A and 9 from B, no match, we move p_A;
- we check 9 from A and 9 from B, we have a match, yield the item and move both p_A and p_B;
- _

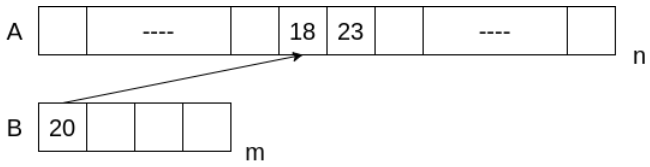
It's an algorithm that runs in $O(n+m)$ time and it's good when $n \approx m$, it's even optimal in that case!

6.1.2 Binary search

Let's suppose that $m << n$ then for each element in B we binary search it in A so the algorithm runs in $O(m \cdot log_2 n)$. It's better when $m \cdot log_2 n < n \implies m < \frac{n}{log_2 n}$.

6.2 Mutual partitioning

A more sophisticate approach is the mutual partitioning: we search in A for the first element in B, when we find it or we find the place in which it should be we search the next items only in the right part of A.



That's good if at each step we splits up the array A which is not always the case, that's because we try to balance the partitioning: instead of starting from $B[0]$ we start from the element in the middle of B and we search the left part of B in the left part of A and the right part of B in the right part of A. So we exploit the cost of single search to short the cost for next searches.

6.2.1 Complexity

Since $|B| = m$ and on average $|B_1| = |B_2| = \frac{m}{2}$ we have #recursive call of $O(log_2 m)$, the recursive relation is:

$$T(n,m) = O(log_2 m) + T\left(n_1, \frac{m}{2}\right) + T\left(n_2, \frac{m}{2}\right)$$

Let's make some assumptions: in the case of fully unbalanced partitioning, which means that in A the element is found totally on the left or totally on the right, we have a good case because we totally drop half of the array B each time, so the recursive calls will be:

$$(n,m) \to \left(n, \frac{m}{2}\right) \to \left(n, \frac{m}{4}\right)$$

So in this case we have a binary search on a fraction of B and we have $log_2 m$ steps so the complexity in that case is $O(log_2 n \cdot log_2 m)$.

The worst case is when we have fully balanced splitting:

$$T(n,m) = O(log_2 m) + T(\frac{n}{2}, \frac{m}{2}) + T(\frac{n}{2}, \frac{m}{2})$$

whose solution is:

$$O\left(m \cdot \left(1 + log_2 \frac{n}{m}\right)\right)$$

which can be proved as the optimal solution in the comparison based model for this problem.

Some observations

If $n \approx m$ then $\frac{n}{m}$ is a constant value, so:

$$O(m \cdot (1 + c)) = O(m)$$

so it is basically merge-based algorithm but in this case merge is better because it uses scan which is better in terms of I/O in respect to random jumping in binary search.

If $m << n$ then the algorithm is:

$$O(m \cdot log_2 n)$$

which basically is binary search.

So this is kinda adaptive to the relation between m and n.

6.2.2 Proof of lower bound

In the comparison based model we can use the decision tree technique to calculate the lower bound, and it is given by:

$$\Omega(log_2 s(m))$$

in which $s(m)$ is the #solutions to the problem.

In the case of the intersection problem the number of solution is:

$$\binom{n}{m}$$

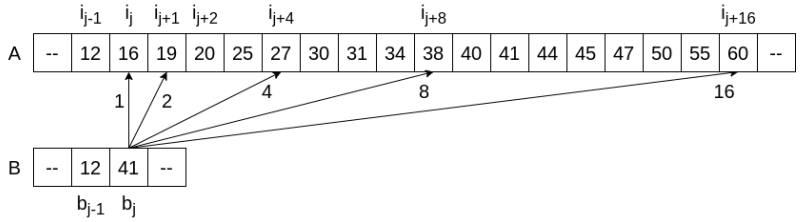
so the lower bound is:

$$log_2 \binom{n}{m} = m \cdot \left(1 + log_2 \frac{n}{m}\right)$$

so that's the proof that mutual partitioning reaches the lower bound for the comparison based model but of course in I/O model it's not since we are jumping around in the memory.

6.3 Doubling search

Doubling search (also known as exponential search or galloping search) states that:



we go from left to right in order to exploit memory pre-fetch, so we compare 41 with 16, since there is no match and 41 is greater than 16 we go to the next element, we compare 41 to 19, no match here too so we double the distance and go to 27, no match neither here and so we double again the distance going to 38, no match here too, double again the distance and we arrive comparing 41 to 60, no match here too but since 60 is greater than 41 we need to search better, but only in the range between i_{j+8} and i_{j+16}.

At each step we double the distance and in the end we will perform a binary search but only on a sub-set, and at each step we reduce the elements we need to check next time! Moreover we restrict search to an area which has the same size of the area we've discarded, that's very powerful!

6.3.1 Complexity

At each step we do a binary search over $A[i_{j-1} + 2^{k-1}, i_{j-1} + 2^k]$ because:

$$A[i_{j-1} + 2^{k-1}] \leq b_j \leq A[i_{j-1} + 2^k]$$

let's call:

$$\Delta_j = min\{2^{k-1}, n\} \leq 2^k$$

which is the total size of the elements seen at each step. Moreover we can state that:

$$2^{k-1} \leq i_j - i_{j-1} \leq 2^k$$

which can be proved by watching at the picture above, so joining the two relations above:

$$2^{k-1} \leq i_j - i_{j-1} \implies 2^k \leq 2(i_j - i_{j-1}) \implies \Delta_j < 2(i_j - i_{j-1})$$

Since sometimes the chunks are overlapped we would like to know how much the overhead can be:

$$\sum_{j=1}^{m} \Delta_j < \sum_{j=1}^{m} 2(i_j - i_{j-1})$$

that's a telescopic sum:

$$= 2\sum_{j=1}^{m} i_j - i_{j-1} = 2[(i_1 - i_0) + (i_2 - i_1) + \_ + (i_m - i_{m-1})] = 2[i_m - i_0] = 2i_m \leq 2n$$

so at the end we have at most $O(n)$ as overlapping part!

Time complexity of each step depends on the jumping part and the binary search one:

$$k - 1 + log_2\Delta_j \leq k - 1 + log_2 2^k = k - 1 + k \approx O(k)$$

in total:

$$\sum_{j=1}^{m} O(k) = \sum_{j=1}^{m} O(log_2\Delta_j) = O\left(\sum_{j=1}^{m} log_2\Delta_j\right)$$

we can use the Jensen's inequality:

$$\sum_{i=1}^{n} log_2 x_i \leq n \cdot log_2 \frac{\sum_{i=1}^{n} x_i}{n}$$

so we have that:

$$O\left(\sum_{j=1}^{m} log_2 \Delta_j\right) \leq O\left(m \cdot log_2 \frac{\sum_{j=1}^{m} log_2 \Delta_j}{m}\right) \leq O\left(m \cdot log_2 \frac{2n}{m}\right)$$

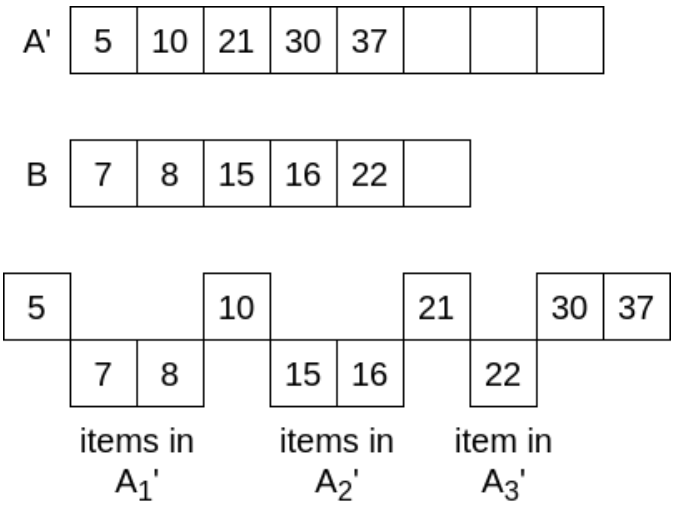
$$= O\left(m \cdot \left(log_2 2 + log_2 \frac{n}{m}\right)\right) = O\left(m \cdot \left(1 + log_2 \frac{n}{m}\right)\right)$$

So we are reaching the lower bound but again the binary search part is a problem whenever we are taking into account the I/O number!

6.4 Two-level approach

It's an algorithm which promotes scan. Given the two sequences A and B we pre-process A by splitting the list logically in some blocks of size L, then we peek the first item of each partition creating A'. So now we have $|A'| = \frac{n}{L}$ and $A[i \cdot L + 1] = A'[i]$.

Once we have this pre-processing we can execute the actual query $A \cap B$. First we need to distribute B among the various items of A': we want to know in which of the blocks A_i falls b_j. We can of course use a merge approach to do it for each of the elements in B.



As we can see if there are no items between two items from A' we can drop a whole A block.

This algorithm then splits-up array B in single blocks:

$$B_i = \{b \in B : A[i \cdot L + 1] \leq b < A[(i+1) \cdot L]\}$$

this step costs:

$$|A'| + |B| = O\left(\frac{n}{L} + m\right)$$

and we get B_i and A_i. Of course B_i could even be empty but overall $B = \bigcup_i B_i$ so $\sum_{i=1}^{\frac{n}{L}} |B_i| = m$, and $|A_i| = L$.

So now we have restricted the intersection problem to single partitions A_i to B_i. With this strategy we can skip L items with just two comparisons, when using the pre-processed lists!

Now we compute $A_i \cap B_i \ \forall i = 1, 2, \_, \frac{n}{L}$ iff $|B_i| \neq 0$ in a merge-based way to promote scan. The cost of this step is $O(|A_i| + |B_i|)$ so:

$$\sum_{i=1, B_i \neq 0}^{\frac{n}{L}} |A_i| + |B_i| = \sum_{i=1, B_i \neq 0}^{\frac{n}{L}} |A_i| + \sum_{i=1, B_i \neq 0}^{\frac{n}{L}} |B_i|$$

in which the first term can be upper bounded by $m \cdot L$ (because we will scan at most m chunks from A, each containing L elements) in the worst case and the second one by m so:

$$= O(m \cdot L + m)$$

NB: another approach could be to shuffle elements using a reversible permutation and then executing the intersection. That's useful to improve storage because we can store the distance between the elements instead of the real ones.

6.4.1 More on 2-level: Interpolation search

The approach of 2-level memory can be used in a wide variety of solutions, for example we can speed up the search for a key over integers using the interpolation search. Take the list of elements $X[1, n] = x_1, x_2, \_, x_n$, we calculate $b = \frac{x_n - x_1 + 1}{n}$, it tells us the range of elements that should be inside a single bucket. Then we build n buckets of size b.

For example:

1	2	3	8	9	17	19	20	28	30	32	36

Once we've splitted the array in blocks we need to create a structure to store bucket starts and end:

- 1, 3: start and end of first block;
- 0, 0: no indices since the block is empty;

- 4, 5: start and end of third block;
- _

Query

To search for an item we first need to calculate the index of the block:

$$i = \lceil \frac{y - x_1}{b} \rceil + 1$$

Once we've found the correct block we just need to binary search in it or do a scan. Supposing of using a binary search the overall cost would be: $O(1+log_2 b)$, but of course the overall complexity is: $O(1+log_2 min\{b, n\})$ because it depends on how many items ends up in a bucket, the key space, etc.

This complexity doesn't take into account the distribution of elements, let's define:

$$\Delta = \frac{max_i x_i - x_{i-1}}{min_i x_i - x_{i-1}} \geq 1$$

it's the ratio between the maximum and the minimum gap between two adjacent items. Then since the maximum is always larger or equal than the average we can state:

$$max_i x_i - x_{i-1} \geq \frac{\sum_{i=2}^{n} x_i - x_{i-1}}{n-1} = \frac{x_n - x_1}{n-1} \geq \frac{x_n - x_1 + 1}{n} = b$$

Then we can estimate the size of a bucket:

$$|I_i| \leq n|I_i| \leq b$$

but we want something better:

$$|I_i| \leq \frac{b}{min_i x_i - x_{i-1}}$$

because inside a block there can be more of the minimum gap, then each b can be upper bounded by the maximum gap:

$$|I_i| \leq \frac{max_i x_i - x_{i-1}}{min_i x_i - x_{x-1}} = \Delta$$

so in each bucket we have at most Δ items.

So our complexity is at most $O(1 + log_2 \Delta)$ time.

Can be proved that if X is evenly distributed (so elements are spaced by the same amount) and we have #universe $= U$ and #items $= n$ then $\Delta = polylog(n)$ with high probability, so:

$$O(log_2 \Delta) = O(log_2 log_2^{\alpha} n) = O(\alpha log_2 log_2 n) = O(log_2 log_2 n)$$

Chapter 7

Hashing

We use hashing in the dictionary problem. A dictionary is a collection of objects: $D = \{O_1, O_2, \_, O_n\} \subseteq U$ composed by $< key, satellite\_data >$. Over this collection we want:

- $search(k)$ to check if $k \in D$;
- $insert(O)$;
- $delete(k)$.

NB: search is exact search, so if we search for x we are looking for an exact match, not a prefix, not search with error. It's also called *membership query*.

7.1 Hash function

We define an hash function as:

$$h : U \to [m]$$

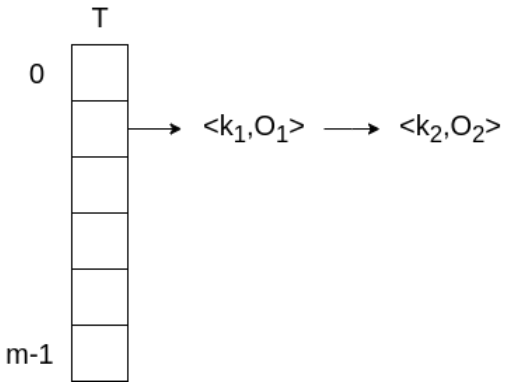
with $[m] = \{0, 1, \_, m-1\}$ with $m << |U|$. So h takes $log_2|U|$ bits and returns $log_2 m$ bits, and the result is called a fingerprint or a digest, that's why we impose $m, n << |U|$.

NB: if U is small there is no need for hash table, we can just use a Direct Access Table which is a table with the same key as the index.

7.2 Classic approaches

7.2.1 Hashing with chaining

We use an array of size m and each cell is a pointer to a linked list, each list contains some entries of the hash table. Whenever we want to add an object we get it's digest and use it to index the array, then we add the entry to the corresponding linked list. The search is of course just a lookup for the digest and a list traversal, deletion is a search and a deletion in the linked list. Basically we use a direct access table and use a linked list to deal with the hash collisions.



The total space is then $O(m+n)$ plus of course the satellite data, so in a more formal way:

$$m \cdot log_2 m + n \cdot [L + 2 \cdot 64] + \text{satellite data}$$

in which $log_2 m$ is the size of a pointer to an item, L is the size of a key and $2 \cdot 64$ is the size of the forward and backward pointer of the linked list.

We want to upper bound the access time, so we need something to estimate the number of elements inside each list, let's use the assumption of *simple uniform hash*:

$$\mathbb{P}(h(k) = i) = \frac{1}{m}$$

so:

$$\mathbb{E}[search] = 1 + \mathbb{E}[len(list)] = 1 + \sum_{k \in D} 1 \cdot \mathbb{P}(h(k) = i) = 1 + \sum_{k \in D} 1 \cdot \frac{1}{m} = \frac{n}{m}$$

We will call $\alpha = \frac{n}{m}$ the *loading factor* of the hash table.

7.2.2 Global rebuilding technique

Another approach which exploits the *amortised argument* states that after some insertions we need to rebuild the entire table, for example:

- we start with $n = n_0$ elements, so we create an hash table with $m_0 = 2 \cdot n_0$ as a size;
- we then have some insertions;
- whenever we hit $n = 2 \cdot n_0$ we rebuild the entire table with $m_1 = 4 \cdot n_0$.

Of course if we delete enough item we rebuild the table, too in order to reduce the memory consumption!

Let's estimate the cost of an insertion between the two builds:

$$1 + \frac{n}{m_0} = 1 + \frac{n}{2 \cdot n_0} \leq 1 + \frac{n_0}{2 \cdot n_0} = 1 + \frac{1}{2} = O(1)$$

so the total cost of inserting phase is $\Theta(n_0)$. Since we've paid $\Theta(n_0)$ to execute n_0 operations we can state that the amortized cost of each insertion is:

$$\frac{\Theta(n_0)}{n_0} = \Theta(1)$$

7.2.3 Problem of simple uniform hashing

In order to be *simple uniform hash* we need that $\exists h \in \mathbb{H}$ such that it can map every key in U in every slot of the hash table.

Let's compute the possible number of mappings to count how many functions corresponds to the definition:

- for k_1 we need to have a mapping for 0, 1, 2, _, $m-1$;
- for k_2 we need to have a mapping for 0, 1, 2, _, $m-1$;
- _
- for k_U we need to have a mapping for 0, 1, 2, _, $m-1$.

Reducing to 2 keys we have m^2 mappings, enlarging to U keys we have m^U mappings, let's call that set $\mathbb{H}$. In order to represent each $h \in \mathbb{H}$ we need $\geq log_s m^U = U \cdot log_2 m$ bits, which is impossible in computational terms!

7.3 Universal hashing

We define the universal hashing as:

$$\mathbb{H} = \{h : U \to [m] : \forall x, y \in U, x \neq y : \#H[h(x) = h(y)] \leq \frac{\mathbb{H}}{m}\}$$

so: fixed a pair of different keys $< x, y >$ the number of functions for which there is a collsion is a fraction of $\mathbb{H}$.

Assuming that hypothesis the probability of picking a function for which there is a collision is:

$$\frac{\frac{|\mathbb{H}|}{m}}{|\mathbb{H}|} = \frac{1}{m}$$

7.3.1 Cost of hashing with chaining and universal hashing

$$\mathbb{E}[len(list)] = \sum_{k' \in D} 1 \cdot \mathbb{P}(h(k) = h(k')) = \sum_{k' \in D} \frac{1}{m} = \frac{|D|}{m} = \frac{n}{m} = \alpha$$

with h chosen at random, we have the same result as before but with a better property since can be effectively used.

7.3.2 Example of universal hashing, with proof

We want to build $h_a : U \to [m]$ with $a \neq 0$ and m prime. To do it we start from a generic key $k \in U$ and we split it in r blocks such that each block is of size $log_2 m$ bits so:

$$r = \frac{log_2 |U|}{log_2 m}$$

We choose $a \in U$ at random but $a \neq 0$ so:

$$\mathbb{H} = \#a \text{ with } a \neq 0 = |U| - 1$$

and we split it in r blocks too. Then we define:

$$h_a(k) = \sum_{i=0}^{r-1} a_i \cdot k_i \mod m$$

That can be seen as scalar product between two vectors.

Proof of universal hashing

Let's fix $x \neq y \in U$ we want to show that:

$$\#h_a : [h_a(x) = h_a(y)] \leq \frac{|\mathbb{H}|}{m}$$

which is equivalent to the universal hashing function.

Let's estimate the $\#a$ such that:

$$\sum_{i=0}^{r-1} a_i \cdot x_i = \sum_{i=0}^{r-1} a_i \cdot y_i \mod m$$

since $x \neq y$ at least a block of x and y is different, let's suppose that the different blocks are x_0 and y_0:

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

since $x_0 - y_0$ is not zero and we have a prime modulus there exists an inverse for it:

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

so in order to have a collision a_0 must be chosen according to the relation above, but of course $a_1, \_, a_{r-1}$ are free to be choosen.

So the values of a are the different ways we can choose $a_1, \_, a_{r-1}$ which is $m^{r-1} - 1$ (-1 because we don't count the 0 combination) so:

$$m^{r-1} - 1 = \frac{m^r}{m} - 1 = \frac{|U|}{m} - 1 \leq \frac{|U| - 1}{m} = \frac{|\mathbb{H}|}{m}$$

NB: of course the usage of m prime is not so good for our uses, we would like to have something in the form of 2^x because we want fast operations like shift and bitmasks, so we can use:

$$h_a(k) = \left[\sum_{i=0}^{r-1} a_i \cdot k_i \mod p \right] \mod m$$

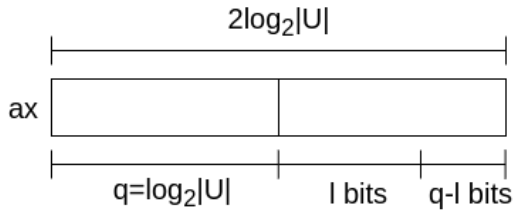
with p the prime number and m a power of 2.

7.3.3 Another universal hashing

We would like to save computational speed avoiding multiplication and divisions. Let's assume $|U| = 2^q$, $m = 2^l$, a odd, we define:

$$\mathbb{H}_{q,l} = \{h_a(x) = (ax \mod 2^q) div 2^{q-l}\}$$

a and 2^q are coprime, so we have a shuffling. Since $|a| = log_2|U|$ and $|x| = log_2|U|$ then $|a \cdot x| = 2log_2|U|$, we take the last $log_2|U|$ bits of $a \cdot x$, then $q = log_2|U|$ and $l = log_2 m$ so in the end we got the upper part of $a \cdot x \mod 2^q$:



So basically we compute a multiplication and keep the middle bits.

7.4 d-left hashing

It's used to remove the number of pointers in order to save space. We start from a table of size m and split it into bins of b slots, so we have $\frac{m}{b}$ bins. We use d different hashing functions and for each key we compute d hashes, access those bins and we place the element inside the most empty to store the object!

Search is constant because we compute hashes and scan the bins:

$$\mathbb{E}[\text{length of the longest list}] = \frac{log_2 log_2 n}{log_2 d} + O(1)$$

when $d \geq 2$.

So if we increase the number of hashing functions we improve the length of bins by very little (it's logarithm factor) but the time complexity increases linearly so the best choice is when $d = 2$.

7.4.1 Power of two choices

It's important to note that in $d = 1$, which is basically the hash with chaining we have:

$$\mathbb{E}[\text{length of the longest list}] = \frac{log_2 n}{log_2 log_2 n}$$

so moving from $d = 1$ to $d = 2$ we have an exponential improvement in length of lists.

7.5 Cuckoo hashing

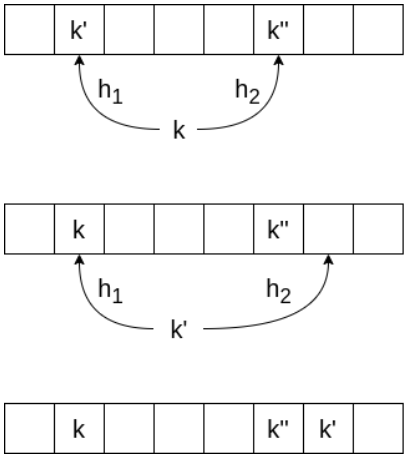
It's a sort of perfect hashing because provides us:

- search in $O(1)$ worst case;
- deletion in $O(1)$ worst case;

- insertion in $O(1)$ amortized expected time (which means that on average the cost is spread among different insertions).

We start from an array of m cells with $m \geq n$, we use two hash functions h_1 and h_2 to access table to check the bins:

- if one of them is empty we put the value inside it;
- if both are filled we put the key into one of them and we relocate the key we've removed:



7.5.1 Theorem 1

$$\forall i, j, \forall c > 1, m \geq 2cn \implies \mathbb{P}(\text{shortest path from i to j is of length L}) \leq \frac{1}{mc^L}$$

We use the concept of path because we can see the cells as nodes of a graph and as arcs we connect the two nodes that can be reached from the key k, so:

- #nodes = m = #cells;
- #edges = n = #keys.

path is important because it states how many jumps we can do during relocations.

It's important to notice that:

$$\alpha = \frac{n}{m} \leq \frac{\frac{m}{2c}}{m} = \frac{1}{2c} < \frac{1}{2}$$

so it means that the hash table is half filled which is not so good for space occupancy. Let's prove the theorem in that case by induction on L:

- $L = 1$:

  $\mathbb{P}$(shortest path from i to j is of length 1) = $\mathbb{P}$(there is an edge from i to j) =

  $$= \mathbb{P}(\exists k : (h_1(k) = i \wedge h_2(k) = j) \vee (h_1(k) = j \wedge h_2(k) = i))$$

  applying the union bound:

  $$\leq \sum_{k \in D} \frac{2}{m^2} = \frac{2n}{m^2}$$

  $m \geq 2cn \implies 2n \leq \frac{m}{c}$:

  $$\leq \frac{\frac{m}{c}}{m^2} = \frac{1}{cm}$$

- assuming that for $L - 1$ is true let's prove it for L:

  $$\mathbb{P}(\text{shortest path from i to j is of length } L)$$

  $$= \mathbb{P}(\exists \text{path of length } L - 1 \text{ from i to z} \wedge \text{path from } z \text{ to } j \text{ of length 1})$$

  applying the union bound:

  $$\leq \sum_z \mathbb{P}(\text{there is a path from i to z of length } L - 1 \wedge \text{ there is an edge from z to j})$$

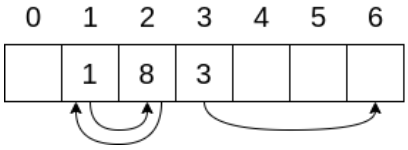
  applying the inductive hypothesis:

  $$= \sum_z \frac{1}{mc^{L-1}} \cdot \frac{1}{mc} = \sum_z \frac{1}{m^2 \cdot c^L} = m \cdot \frac{1}{m^2 \cdot c^L} = \frac{1}{m \cdot c^L}$$

7.5.2 When cuckoo hashing fails

Let's use $h_1(x) = x \mod 7$, $h_2(x) = 2x \mod 7$, $m = 7$ and let's insert 1, 3, 8, 15:

- 1 can go both into $h_1(1) = 1$ and $h_2(1) = 2$, we insert it into 1;
- 3 can go both into $h_1(3) = 3$ and $h_2(3) = 6$, we insert it into 3;
- 8 can only go into $h_2(8) = 2$ since $h_1(8) = 1$ is already filled;
- 15 neither can go into $h_1(15) = 1$, nor $h_2(15) = 2$, so we need to relocate something, let's check the graph;



  we have a cycle between 1 and 2 and we want to insert something in one of the two, so the insertion fails!

NB: just one cycle is not enough to fail an insertion, what we need for failing is two cycles and an insertion which joins them!

7.5.3 Failing

$$\mathbb{P}(failing) \leq \mathbb{P}(\text{exists 2 cycles}) \leq \mathbb{P}(\exists \text{cycles}) =$$

$$\mathbb{P}(\exists z, L : \text{there is a path from z to z of length L})$$

applying union bound:

$$\leq \sum_{z \in T} \sum_{L \geq 1} \frac{1}{mc^L} = \sum_{z \in T} \frac{1}{m} \sum_{L \geq 1} \left(\frac{1}{c}\right)^L$$

since $c > 1$ it's a known series:

$$= \sum_{z \in T} \frac{1}{m} \cdot \frac{1}{c-1} = \frac{1}{c-1}$$

7.5.4 Amortized cost in insertion

Fixing $c = 3$ we have:

$$\mathbb{P}(\text{insertion will fail}) \leq \frac{1}{2}$$

$$m \geq 6n \implies \alpha = \frac{n}{m} \leq \frac{1}{6}$$

which are not so nice.

So we have a huge table and half of insertions will fail, but let's check the amortized expected cost:

- we start supposing of having $n_0 + \epsilon n_0$ keys, so we will build a table with $m_0 = 6(1+\epsilon)n_0$ but having only n_0 actual keys;
- after ϵn_0 insertions we have filled the available space so $m_0 = 6 \cdot \#\text{keys}$.

Since $m \geq 6n$ the probability of insertion fails is $\leq \frac{1}{2}$.

We sit at the end of the insertions and asks us the probability of failing over the past $(1+\epsilon)n_0$ insertions, it's $\frac{1}{2}$ so we do at most 2 constructions before achieving that amount of insertions (which means we redraw h_1 and h_2).

Since construction cost is linear if we distribute it over all the n_0 insertions we have amortized cost. So:

$$\text{if } c \leq 2, \forall x, y \mathbb{P}(\text{there is a path from x to y}) = \frac{1}{m}$$

which means that x and y collides:

$$= \mathbb{P}(\exists L : \text{exists a path of length L from x to y})$$

applying the union bound:

$$\leq \sum_{L \geq 1} \frac{1}{mc^L} = \frac{1}{m} \sum_{L \geq 1} \frac{1}{c^L}$$

since $c > 1$ it's a known series:

$$= \frac{1}{m} \cdot \frac{1}{c-1}$$

for $c = 2$ it's $\frac{1}{m}$ and for $c > 2$ it's $\leq \frac{1}{m}$, so:

$$\mathbb{P}(collision) \leq \frac{1}{m}$$

So average #keys colliding with some keys k is $\frac{n}{m} \leq 1$, so the average insertion cost is $O(1)$ time expected.

But the space occupancy continues to be bad!

In practice

In practice we mix d-left and cuckoo hashing so instead of single cells we use buckets. It guarantees 95% of occupancy!

7.6 Minimal Ordered Perfect Hash Function

7.6.1 Perfect hash function

h is a perfect hash iff:

$$\forall k_1, k_2 \in D, k_1 \neq k_2 \implies h(k_1) \neq h(k_2)$$

basically: no collisions among the elements in the dictionary. That's useful because it guarantees direct access to the elements.

7.6.2 Minimal Perfect hash function

A perfect hash function is minimal iff:

$$m = n$$

we have as many positions in the table as object to store.

7.6.3 Minimal Ordered Perfect hash function

A minimal perfect has function is ordered if:

$$\forall k_1, k_2 \in D : k_1 < k_2 \implies h(k_1) < h(k_2)$$

so we can say that $h(k)$ is the rank of k in D.

An hash of this type can be useful in case of database when we create a dictionary with direct access for elements that are already inside.

7.6.4 Designing a MOPHF

Take the following strings sorted as a key, fix the value for $h(t)$ for each of them:

k	$h(t)$
abacus	0
cat	1
dog	2
flop	3
home	4
house	5
son	6
trip	7
zoo	8

so we have $n = 9$.

Other approaches

We can store those strings with pointers in an array, the space is $O(n+N)$ with:

- n: the pointers;
- N: the strings.

during the search we use a binary search to find out the position, so the cost is $O(p \cdot log_2 n)$ in which:

- p: is the cost for string comparison;
- $log_2 n$: is the cost of the binary search.

We can use a trie data structure with $O(N)$ space and $O(p)$ time. Both those approaches aren't too good because we want:

- space occupancy proportional to n, not to N;
- slower access time than trie.

Building the hash table

We take two perfect hashes h_1 and h_2:

$$h_1, h_2 : U \to [m']$$

with m' as prime number, and get the values of each $h_1(k), h_2(k)$:

$h(t)$	$h_1(k)$	$h_2(k)$
0	1	6
1	7	2
2	5	7
3	4	6
4	1	10
5	0	1
6	8	11
7	11	9
8	5	3

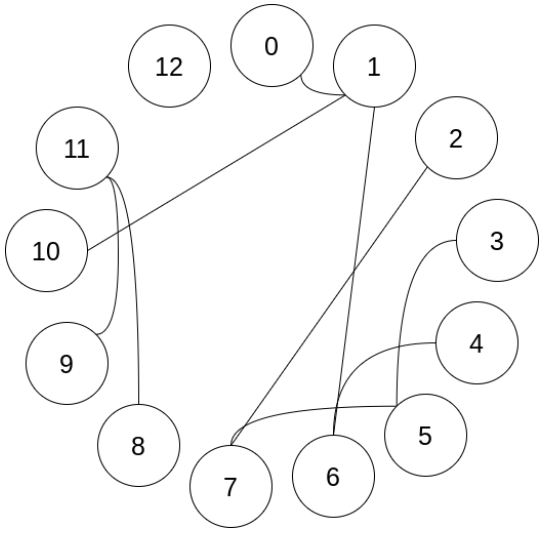
Now we construct an array g which maps the function $g : [m'] \to [n]$, then we use g to build the actual hashing function:

$$h(t) = g(h_1(t)) + g(h_2(t)) \mod n$$

Let's prove that we can build the wanted $g(h)$ with high probability (if not we reshuffle h_1 and h_2 and entirely rebuild the g function).

Building $g(h)$

We start building a graph with m' nodes, then we connects them with edges as following: from $h_1(x)$ to $h_2(x)$.



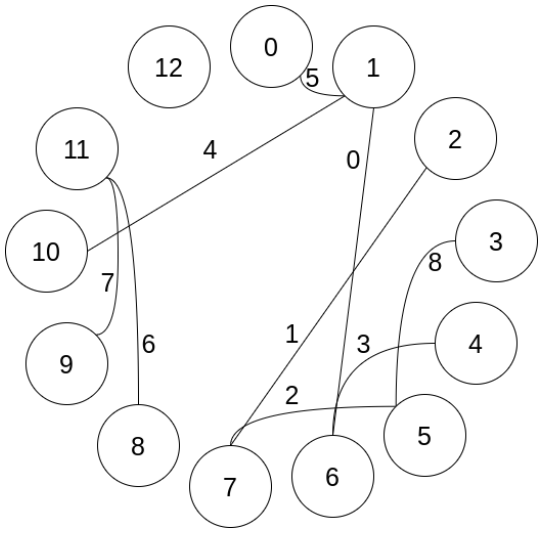
If there is a cycle inside the graph we throw away the structure and recompute h.

Now we can build the equations:

$$h(abacus) = 0 \implies g(h_1(abacus))+g(h_2(abacus)) = 0 \mod 9 \implies g(1)+g(6) = 0 \mod 9$$

at the end we build n equations in m variables (the values of g) that can be solved in a lot of ways. We'll solve it using the graph itself writing the solutions over the arcs in the graph: we can set $g(x)$ as the value we want for one of the nodes, and then from it we navigate the graph solving the equations:

- we set $g(2) = 0$;
- $g(2) + g(7) = 1 \implies g(7) = 1$;
- $g(7) + g(5) = 2 \implies g(5) = 1$;
- $g(5) + g(3) = 8 \implies g(3) = 7$;
- of course when a path is over we start again by picking a node, setting a value for it and then again traversing the graph until all the nodes are done.



To build g we need $O(m)$ time and $O(n)$ space, which is the table for g. To store h_1 and h_2 we can use $ax + b \mod p \mod m'$ because they are universal hashes, so we can build the hash-table without storing the strings.

During the search we hash the string using h_1 and h_2, we access g at those indexes and sum the results.

7.7 Bloom filters

In the context of dictionary problem a bloom filter allows us to:

- $insert(k)$;
- $membership(k)$.

we can't delete a key!

It is a very succint data structure because it does not store the keys, as a drawbacks we can incur into *false positive* errors, so a match means a *maybe*. It's called *one side error* because a yes is a maybe but no is no. Moreover we can control the error (eg make it really small)!

We need:

- r different universal hash function: h_1, h_2, _, h_r:

$$h_i : U \to [m], m > n$$

- a binary array $B[0, m-1]$

7.7.1 Insertion

To insert k we set:

$$B[h_i(k)] = 1 \forall i = 1, 2, \_, r$$

insertion is $O(r)$ time.

7.7.2 Membership

Given a key k we return:

$$yes \iff B[h_i(k)] = 1 \forall i = 1, 2, \_, r$$

7.7.3 Complexity

A bloom filter occupies m bits ($m = r \cdot n$ with $r \approx 30 - 60$) while hashing with chaining/cuckoo hashing uses $m \geq 2cn \cdot log_2 U$ bits, so if the constant multiplied by n is < 60 it's more useful to go with cuckoo hashing.

In modern database B^+-tree is used and each level stores a bloom filter that stops tree visit if it says no, in order to prevent some I/Os.

Let's estimate $\mathbb{P}(B[j] = 0)$: we have n keys and every key have to set r bits, so:

$$\mathbb{P}(B[j] = 0) = \left(\frac{m-1}{m}\right)^{nr} = \left(1 - \frac{1}{m}\right)^{nr} = \left(1 - \frac{1}{m}\right)^{\frac{nr}{m} \cdot m}$$

$$= \left[\left(1 - \frac{1}{m}\right)^{m}\right]^{\frac{rn}{m}} \xrightarrow{\infty} e^{-\frac{rn}{m}}$$

so:

$$\mathbb{P}(B[j] = 1) = 1 - e^{-\frac{rn}{m}}$$

NB: we are supposing that $B[a]$ is independent from $B[b]$ which is not correct but a good approximation.

Let's estimate the error:

$$\mathbb{P}(error) = \mathbb{P}(k \notin D \wedge \forall i = 1, 2, \_, r B[h_i(k)] = 1)$$

$$= \mathbb{P}(B[j_1] = 1 \wedge B[j_2] = 1 \wedge \_ \wedge B[j_r] = 1) \approx \mathbb{P}(B[j] = 1)^r = (1 - e^{-\frac{rn}{m}})^r$$

To study the function we fix two variables and vary the last one, let's fix m and n, we have a function in r. Studying the function we can find that the minimum error is gained for:

$$r_{opt} = \frac{m}{n} ln2$$

which gives us the error:

$$\epsilon_{opt} = \left(\frac{1}{2}\right)^{r_{opt}} = (0.6...)^{\frac{m}{n}}$$

so each bit we add decreases the error by a factor 0.6.

Eg: $m = 2n \to \epsilon_{opt} = 0.38$, $m = 5n \to \epsilon_{opt} = 0.09$, so an error rate of 9%!

Of course the less the error we want, the more the hash function and the size we need, so usually we give away the minimal error and use $r = 20$ with $\epsilon = \left(1 - e^{-\frac{20n}{m}}\right)^{20}$.

We have a lower bound: every data structure which makes an error ϵ over n keys occupying space m must use m bits s.t.:

$$m \geq n \cdot log_2 \frac{1}{\epsilon}$$

For bloom filters in particular (in optimal case):

$$\epsilon = 2^{\frac{m}{n} ln(2)} \implies \frac{1}{\epsilon} = 2^{\frac{m}{n} ln(2)} \implies log_2 \frac{1}{\epsilon} = \frac{m}{n} ln(2)$$

$$\implies m = \frac{1}{ln(2)} \cdot n \cdot log_2 \frac{1}{\epsilon}$$

so we have a factor different from the lower bound.

Example of usage

Two machines in a p2p network wants to synchronize their knowledge, so we have two sets A and B and we want to compute $A \cap B$. Let's suppose that A wants to send hashes of movies: $|A| log_2 U$ bits sent, A builds up BF_A of size m_A ($\epsilon = (0.6)^{\frac{m_A}{n_A}}$) and sends it. So A sends m_A bits. Then B receives BF_A and $\forall b \in B : BF_A(b) =:$

- yes: b maybe $\in A \cap B$: it's yes for sure for $b \in A \cap B$ and is yes for some $b \in B \cap b \notin A$ which are the errors;
- no: $b \notin A$, so $b \notin A \cap B$ and we download it.

So let's assume that $\epsilon = 1\%$:

$$\mathbb{E}[errors] = \#\text{checked elements} \cdot (0.6)^{\frac{m_A}{n_A}} \leq |B| \cdot (0.6)^{\frac{m_A}{n_A}}$$

so we will get $\#yes = |A \cap B| + |B| \cdot (0.6)^{\frac{m_A}{n_A}}$ which is the number of correct plus the number of errors.

7.7.4 Spectral bloom filter

The spectral bloom filter allows us to:

- $counting(k)$;
- $insertion(k)/increasing(k)$;
- $deletion(k)/decreasing(k)$

instead of a binary array we use an array of integers of 32 or 64 bits C, then we use r hash functions $h_1, h_2, \_, h_r : U \to [m]$.

Increment

$$C[h_i(k)] += 1 \forall i = 1, 2, \_, r$$

Decrement

$$C[h_i(k)] -= 1 \forall i = 1, 2, \_, r$$

Counting

$$min_{1 \leq i \leq r} C[h_i(k)]$$

The probability of an error is the same as usual bloom filter:

$$\mathbb{P}(error) = \left(\frac{1}{2}\right)^{r_{opt}} = (0.6)^{\frac{m}{n}}$$

Chapter 8

Prefix search in strings

Of course in this case too we are talking about variable length objects, not just text strings. We store them with an array of pointers of sorted strings. We define N as the total length of strings and $\sigma = |\Sigma|$ as the alphabet size, then we will use \$ as the smallest symbol in the lexicographic order and # as the biggest one.

The problem is: given a pattern $P[1, p]$ we want to find (count or retrieve) all the occurrences of P as prefix of the dictionary of strings. Hashes can't be used since we are searching for a partial part of the key.

Let's suppose that our pattern is $P = te$ we can notice that:

- all the strings that matches the prefix P are contiguous in the array, since it's sorted;
- the starting position is the lexicographic position of $P\$$;
- the ending position is the lexicographic position of $P\#$.

As we are working we can already say that:

- count is the subtraction end to start, and it's $O(1)$ operation;
- retrieve is proportional to the occurrences.

when we already know ending and starting positions. We just need to find those positions.

8.1 Binary search

We can use binary search:

- $O(plog_2 n)$ time;
- $O(\frac{p}{B} log_2 n)$ I/Os;
- $O(Nlog_2\sigma + nlog_2 N)$ bits space.

8.1.1 Improve locality

To boost performance instead of asking to the system to allocate single strings we ask for the whole memory and store strings one after the other. Then instead of an array of pointers we can use an array A of offsets from the start of the array.

Now whenever I restrict to a part of the array A I restrict also on a part of memory, that can save I/Os in retrieve because we can exploit locality of reference. So we achieve:

$$O\left(\frac{P}{B} log_2 \frac{N}{B}\right)$$

I/Os.

Moreover now we exploit the fact that the array is sorted so common prefixes are near inside this version.

8.2 Front coding

Using front coding we can gain a 50% compression on average, it states that you store strings with < length shared prefix, remaining suffix >:

Eg: alcatraz, alcool, alcyone, alu, box: $< 0, alcatraz >, < 3, ool >, \_, < 0, box >$

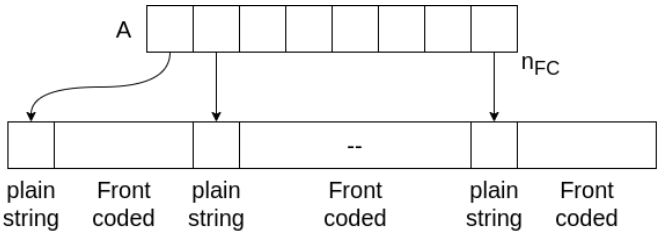
But now how big we create the number to store the length? Engineering the store we use a single byte which can store from 0 to 255, and if the length is more we allocate 2 more bytes to store the excess.

8.2.1 Front coding and 2-level indexing

In prefix search this storing is useless since if we want to random access we should go backward until we find the actual real prefix of the random element we are accessing which could lead us even to the first string.

What we can do to make it useful is to split the strings inside some blocks and front-encode each block. Then on retrieve we start from the first string of the block and go on to rebuild the entire block:

- first string is stored uncompressed for each block;
- we can build A using pointers to the block start.



So in search we have:

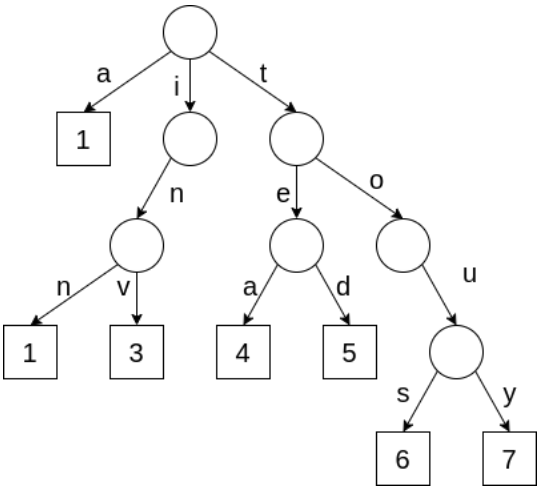
$$\frac{P}{B} log_2 \frac{FC(D)}{B}$$

I/Os for the binary search part, then the scan of each block is $O(1)$ I/Os so only the search in A has weight.

In reality the second phase is $O\left(\frac{FC(occ)}{B}\right)$ I/Os because the matched could traverse different blocks which must be retrieved to return all the items.

8.3 Trie

We can use a trie:



Let's suppose we want to look for $P = te$, so we need to find the starting node, so we search for $te\$$:

- we start from the root;
- we pick t;
- we have e and o, we pick e;
- we have a and d, we pick a since $a \geq \$$, starting point is 4.

Now we need to find the ending point, so let's query for $te\#$:

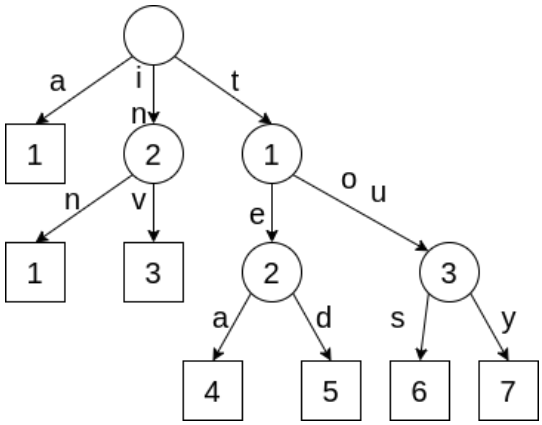
- we start from the root;
- we pick t;
- we have e and o, we pick e;
- we have a and d, we pick d since is the last one $d \leq \#$, ending point is 5.

So we dropped to $O(p)$ time and $O(p)$ I/Os and depending on p it can be useful than the other approaches.

To improve further of course we could build a trie over the starting elements of each block, so always using the two level indexing.

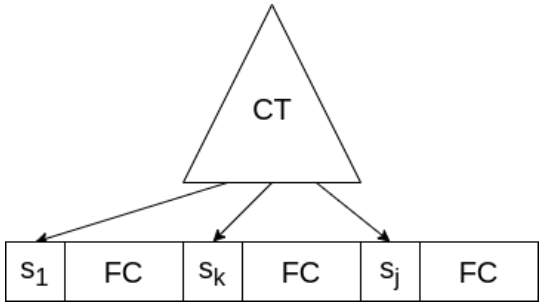
8.3.1 Compacted trie

We can compact a trie deleting the nodes with one outgoing edge and storing the whole substring from that path on the outgoing edges and storing the prefix size on the node:



If we want to avoid the storage of the letters, in case of long substrings we can store on the edges a triple of values: $<$ string number, starting position, end position $>$, so for example instead of *in* we can store $< 2, 3, 3 >$, so each edges has constant size! Now the trie has size: $l + l - 1 + (2l - 1) = O(l) \approx$ #strings, in which l is the number of leaves.

In the end our structure to compute prefix search is:



we need $O(p)$ I/Os at most, which is the trie traverse and the eventual string extraction since edges are compacted.

NB: if the first string of each block fits inside a block we can save I/Os trying to always use it in the tuples of compacted edges.

8.4 Another 2-level index by example

We want to design a two-level index data structure based on compacted tries and blocks of 3 strings. In input:

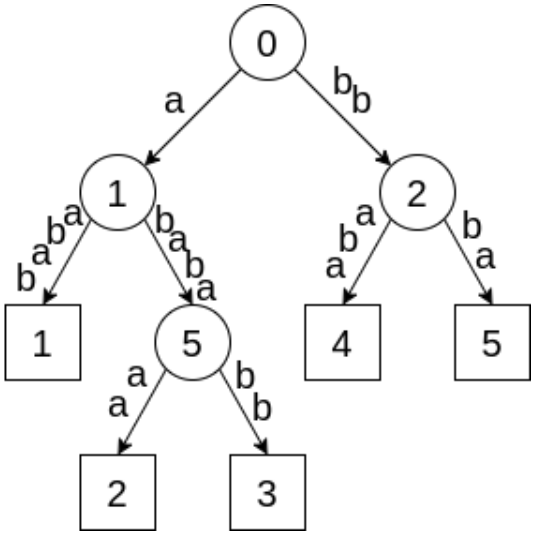
- aabaa, aabab, aabba;
- ababaaa, ababaab, abababa;
- abababb, ababbbb, abbaaaa;

- bbaba, bbabaa, bbabab;
- bbba, bbbab, bbbb.

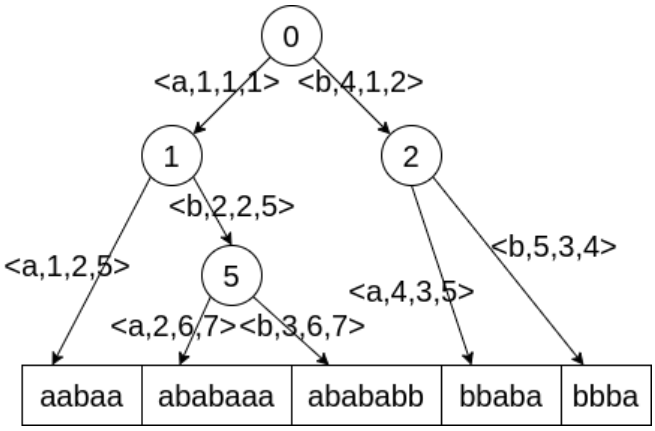
We need to compress strings in block with front coding, first string of each block is kept as is, the other ones will be encoded as $<$ prefix size, postfix $>$:

- $< 0, aabaa >, < 4, b >, < 3, ba >$;
- $< 0, ababaaa >, < 6, b >, < 5, ba >$;
- $< 0, abababb >, < 4, bbb >, < 2, baaaa >$;
- $< 0, bbaba >, < 5, a >, < 5, b >$;
- $< 0, bbba >, < 4, b >, < 3, b >$.

the compressed part goes to the disk and for the first string we build an index, assuming that we have enough space in internal memory to store first string for each block:



we have our uncompacted trie but we can improve the memory occupancy avoiding to store the substrings and replacing them with <character to branch, string index, start position, end position>:



So now each node is constant size which is good if strings are long, as in our case, and moreover we can estimate the occupied space:

- $\#nodes < \#leaves < \frac{n}{B}$ because each node is at least binary;
- $\#leaves = \frac{n}{B}$;
- $\#edges = \#nodes + \#leaves - 1 < 2\frac{n}{B}$.
  NB: -1 because root node has no in-going edges.

Let's search for $P = abbab$:

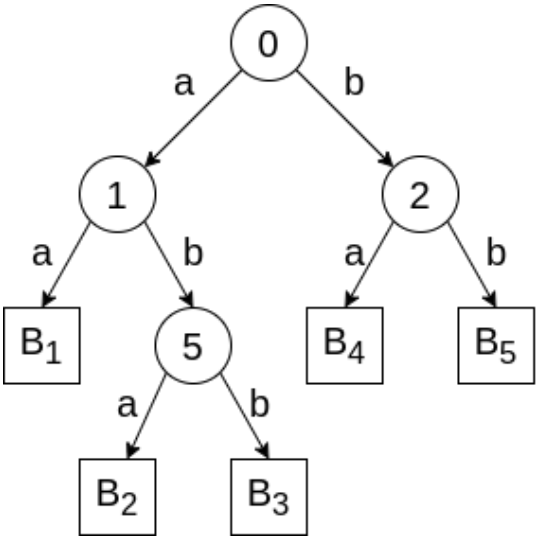
- we start from the root and choose the edge a because the string starts with a;
- we decompress the edge and get a;
- second character is b, we check the node and there is an edge starting with b, so we use it;
- we decompress the edge and get $baba$;
- check for match: $baba \neq bbab$ so we have a mismatch!
- all strings from this edge starts with aba but we are looking for abb so we return iterators from between s_3 and s_4, so to start search in disk we must start from block 3.
- we fetch block 3, we decompress it and start checking for match;
- we find nothing in block 3;
- we know block 4 is already out of scope because it starts with b;
- so no matches!

Whether we fit strings in memory or in disk we have:

- if first string is in memory: no I/Os for index traversal and a single I/O for the block found, then we pay $O(p)$ time comparison for traversing the trie and $O(B)$ time for searching in the block, so in total $O(p + B)$ time;
- if the first string is on the disk: $O(p+1)$ I/Os because we need to decompress each edge and $O(p + B)$ time.

8.5 Patricia trie

A patricia trie is a data structure useful to treat the prefix search problem without taking into account if the strings are in memory or on a disk. The main idea is to drop edge labels from the compacted trie except for the first character, so for example we would have:



so pointers point to the blocks and not to the strings. It's kinda a lossy compression and we search in three phase, let's suppose we are looking for $P = abbab$:

- Down-word traversal:
  - we match a in position 1, so we take the edge;
  - we batch b in position 2, so we take the edge;
  - the pattern is shorter than 6 characters so we don't match. We stop the traverse and pick all the strings from this node.

  Up to now we have no I/Os.

- We pick one of the strings in the result (for example S_2) and compute the longest common prefix:

  $$lcp(S_2, P) = ab$$

  we have a mismatch cause the ending part of P bab is missing. Moreover this information tells us that the pattern is on the right of S_2. We have 1 I/O and $O(p)$ time.

- Up-word traversal: we move the cursor to the next block pointed by the node we have chosen above.

Chapter 9

Substring problem

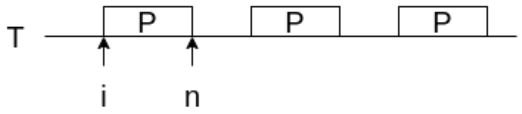
Given a text $T[1,n]$, we want to process it in order to search on-the-fly for $P[1,p]$ and return occurrences of P as a sub-string of T.

NB: on-the-fly means that we have the text (or the dictionary) before and that we will execute different queries, and not that we are given text and pattern together. In that case we just scan.

NB: P is any sequence of symbols (not forced to sequences of bytes).

An important context of usage is Bioinformatics: we search for DNA sequences.

Here too find occurrences means retrieval of positions and counting. To solve efficiently we define $T[i,n]$ as a text suffix from position i to position n:



So we have an occurrence of P in position i if and only if pattern P is prefix of suffix $[i,n]$, so we can reduce the problem to a prefix search! But we need to manage the space better.

Ex: $T = mississippi\$$, $P = si$: P is a prefix of $T[4,12] = sissippi\$$ and $T[7,12] = sippi\$$ so substring search of $T[1,n]$ is reduced on prefix search on $SUFFIX(T)$ which consists of n suffixes of T.

$SUFFIX(T)$ is quadratic: $\sum_{i=1}^{n} i = \Theta(n^2)$ so in practical it is not so interesting.

9.1 Suffix array

It's an array which contains all the the prefixes $T[i,n]$ sorted alphabetically:

S_A	sorted $SUFFIX(T)$
12	\$
11	$i\$$
8	$ippi\$$
5	$issippi\$$
2	$ississippi\$$
1	$mississippi\$$
10	$pi\$$
9	$ppi\$$
7	$sippi\$$
4	$sissippi\$$
6	$ssippi\$$
3	$ssissippi\$$

Of course storing sorted $SUFFIX(T)$ is quadratic but we can just store the S_A and T and use S_A as indirection to T.

So suffix in position j is $T[S_A[j], n]$, then S_A has size $\Theta(nlog_2 n)$ bits and T is $\Theta(nlog_2\sigma)$ bits (using σ as alphabet).

Assuming we have a DNA sequence, then we have $\Sigma = \{A, C, G, T\}, \sigma = 4$, so $T = 2n$ bits and $S_A = nlog_2 n$ bits. Human genomes is $\approx 2^{30}$ bases, so $n = 2^30$ (1GB) so S_A is $2^{30}log_2 2^{30} = 30 \cdot 2^{30}$ (30GB).

To search we execute a binary search using S_A as index to jump to the text, of course we always search for $P\$$ and $P\#$ to determine the range!

So in the worst case substring search costs $O(plog_2 n)$ time.

NB: for each jump in S_A we will have cache miss because S_A are just indices to T. the count can be gathered just in $log_2 n$ jumps but retrieval needs scan in S_A so it's $O(occ)$ and $O(\frac{occ}{B})$ I/Os, if in the end we would like to retrieve the result sorted we'll have: $O(occ \cdot log_2 occ)$.

NB: in comparison model we can achieve substring search in $O(p + log_2 n)$

9.2 Building suffix array

A non-optimal approach (but elegant) is:

```
SA_build(char *T, int n, char **SA){
    for i = 0; i < n ; i++:
        SA[i] = T+i;
    qsort(SA, n, sizeof(char*), suffix_cmp);
}

suffix_cmp(char** p, char** q){
    retur strcmp(*p, *q);
}
```

(It's hermetic code).

It's a cubic algorithm but if quicksort on average is $O(n \cdot nlog_2 n) = O(n^2 \cdot log_2 n)$, which is still quadratic, not taking into account the cache misses for string comparison.

In the worst case building costs:

$$\left(\frac{n}{B}\right) \cdot nlog_2 n$$

I/Os, $n^2 \cdot log_2 n$ time and $n \cdot log_2 n$ extra bits of storage.

NB: there is also a version of this problem which uses prefix tree which uses a trie.

9.3 Longest common prefix

LCP	S_A	sorted $SUFFIX(T)$
0	12	\$
0	11	i\$
1	8	$ippi$\$
1	5	$issippi$\$
4	2	$ississippi$\$
0	1	$mississippi$\$
0	10	pi\$
1	9	ppi\$
2	7	$sippi$\$
1	4	$sissippi$\$
3	6	$ssippi$\$
	3	$ssissippi$\$

$LCP[i]$ is the length of the longest common prefix between $SA[i]$ and $SA[i+1]$, so it is $LCP[0, n-1]$ and in the worst case can be computed in $O(n^2)$.

It's useful for queries like:

- Does it exists a substring of T that repeats (so ≥ 2) and has length L? Can be solved using brute-force in $O(n(nL)) = O(n^2)$ but using LCP array we just need to scan for a value $\geq L$ and if there is at least one we return *yes*, otherwise *no*. It's a scan over LCP so it is $O(n)$ time and $O\left(\frac{n}{B}\right)$ I/Os.

- Find the longest repeated substring. Using LCP we just need to find max in LCP which is a scan so $O(n)$ time and $O\left(\frac{n}{B}\right)$.

- Check whether exists a substring of length at least L that repeats at least c times. Using LCP we look in the array at for at least L and count how many successives are at least L, if count is at least $c-1$ we say *yes*, otherwise *no*. Basically we are looking for a subarray $LCP[i, i+c-q]$ s.t. each elements $\geq L$.

Chapter 10

Data compression

10.1 On data compression: Entropy of a source

Let's imagine a source which emits symbols from the alphabet A with some frequency, we define it's *entropy* with:

$$H_0 = \sum_{\sigma \in A} p_\sigma log_2 \frac{1}{p_\sigma}$$

in which:

- p_σ is the probability of emitting the symbol σ, so $0 \leq p_\sigma \leq 1$;
- $H_0 \geq 0$ because $log_2 \frac{1}{p_\sigma} \geq 0$;
- 0 in H_0 stands for entropy of single symbol;
- the unit of measure is *bits*.

H_0 is a convex function and it's maximum is when all probabilities are equals, so if:

$$p_\sigma = \frac{1}{|A|}$$

so:

$$H_0 = \sum_{\sigma \in A} p_\sigma log_2 \frac{1}{p_\sigma} \leq \sum_{\sigma \in A} \left(\frac{1}{|A|} \right) log_2 \frac{1}{\frac{1}{|A|}} = |A| \cdot \left(\frac{1}{|A|} \right) log_2 |A| = log_2 |A|$$

So we got a higher bound for the entropy of a source.

Let's try to give an interpretation for that formula: suppose that X is a random variable that assumes the value $log_2 \frac{1}{p_\sigma}$ with probability p_σ, then:

$$\mathbb{E}[X] = \sum_{\sigma \in A} p_\sigma \cdot \left(log_2 \frac{1}{p_\sigma} \right) =$$

We can interpret entropy as the average of a source emitting symbol $log_2 \frac{1}{p_\sigma}$ with probability p_σ, so if $log_2 \frac{1}{p_\sigma}$ is the information content of the symbol σ then H_0 is the average of information content.

Ex: if $p_A = 1$ and $p_B = 0$ then:

- $p_A \cdot log_2 \frac{1}{p_A} = 0$: if the source always sends A then there is no content;
- $p_B \cdot log_2 \frac{1}{p_B} \to \infty$: if the source says a very rare symbol we gather a lot of information.

10.1.1 Shannon's theorem

Any *prefix-free code* for the source S takes an *average* of bits per symbol $\geq H_0$, so H_0 is a lower bound.

A prefix-free code is an encoding in which there is no symbol which is a prefix of another symbol.

Ex: $a = 0, b = 01, c = 1$: we have that a is prefix of b, so if we read the bits 01 we don't know if it's the encoding of b or ac.

$a = 0, b = 10, c = 111$ is prefix-free, let's calculate the average code length with $p_a = \frac{1}{4}, p_b = \frac{1}{2}, p_c = \frac{1}{4}$:

$$\mathbb{E}[|cw(\sigma)|] = \sum_{\sigma \in A} p_\sigma \cdot |cw(\sigma)| = \frac{1}{4} \cdot 1 + \frac{1}{2} \cdot 2 + \frac{1}{4} \cdot 3 = \frac{1}{4} + 1 + \frac{3}{4} = 2 \frac{bits}{symbol}$$

with $cw(\sigma)$ the codeword of σ, so the encoding. Is it a good encoding or not? Let's check the lower bound:

$$\frac{1}{4} log_2(4) + \frac{1}{2} log_2(2) + \frac{1}{4} log_2(4) = \frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 = \frac{1}{2} + \frac{1}{2} + \frac{1}{2} = \frac{3}{2} = 1.5 \frac{bits}{symbol}$$

So the code we used is not optimal since it's higher than the lower bound. An optimal encoding could be: $a = 00, b = 1, c = 01$:

$$\mathbb{E}[|cw(\sigma)|] = \frac{1}{4} \cdot 2 + \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 = \frac{1}{2} + \frac{1}{2} + \frac{1}{2} = \frac{3}{2} = 1.5 \frac{bits}{symbol}$$

10.1.2 Golden rule of data compression

$$p_\sigma > p_{\sigma'} \implies |cw(\sigma)| \leq |cw(\sigma')|$$

the bigger the probability, the shorter the symbol encoding.

Who gives us the probabilities and the encoding? If the message T is known we can count the appearances of each element and we can use:

$$p_\sigma \approx freq(\sigma) = \frac{occ(\sigma)}{|T|}$$

this model is called *semi-static model*.

There is also a *dynamic model* in which we start with each symbol with equal probability and at each new emitted symbol we adjust those probabilities. We use this technique in streaming model and each time we change the encoding.

10.2 Integer coding

Given $S = s_1, s_2, s_3, \_, s_n$ with $s_i \in \mathbb{N}$ and $s_i < s_{i+1}$ we define the *gap sequence* as:

$$S' = s_1, (s_2 - s_1), (s_3 - s_2), \_$$

NB: the property $s_i < s_{i+1}$ doesn't lack of generality because we can always find a transformation from every S to a sequence with this property, for example we can use the *sum sequence* in which each element is the sum of all the previous one.

NB: $s_i \in \mathbb{N}$ doesn't lack of generality because we can exploit the numerability of $\mathbb{Z}$ to create a map $\mathbb{Z} \to \mathbb{N}$:

$$\begin{cases} 2x & \text{if } x \geq 0 \\ 2|x| + 1 & \text{if } x < 0 \end{cases}$$

So our problem is: how to represent S with the least number of bits?

Es: $S = 3, 5, 6 \implies S' = 3, 2, 1$, from S to S' ad vice versa is straightforward.

Now that S' has some repetitive element we can exploit those repetitions to compress data.

10.2.1 Naive encoding

Given $S = s_1, s_2, \_, s_n$ with $s_i < s_{i+1}$, we pick $S^* = s_n + 1$ we encode each s_i in a fixed-len of $\lceil log_2 S^* \rceil$ bits. It's simple but it's a waste of space.

10.2.2 Unary encoding

Given $x \geq 1$ we define:

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

Ex: $U(1) = 1, U(2) = 01, U(3) = 001$.

Optimal distribution

This approach is stupid but it's optimal for some distributions: when $|cw(\sigma)| = log_2 \frac{1}{p_\sigma}$, it happens when:

$$|U(\sigma)| = log_2 \frac{1}{p_\sigma} \implies \sigma = log_2 \frac{1}{p_\sigma} \implies 2^\sigma = \frac{1}{p_\sigma} \implies p_\sigma = \frac{1}{2^\sigma}$$

So if distribution of σ is exponentially decreasing unary code is optimal!

NB: we used Shannon's theorem to derive the distribution which optimize our encoding.

10.2.3 γ-code

Given $x \geq 1$:

$$\gamma(x) = 0^{l-1} bin(x)$$

with $l = |bin(x)| = \lceil log_2(x + 1) \rceil$ and $bin(x)$ the binary encoding.

Ex: $\gamma(9) = 0001001$

It works because each x starts with a 1 for sure (since $x \geq 1$), so $|\gamma(x)| = 2l - 1 = 2\lceil log_2(x+1) \rceil - 1 \approx 2log_2 x - 1$. So $\gamma(x)$ is twice as long as x but it's structure is useful because counting zeroes we can know how to decode: for example 0010100111010... we parse it:

- 00: $len(00) = l - 1$ so the number is long 3 bits;
- 101: the number is 5;
- 00: $len(00) = l - 1$ so the number is long 3 bits;
- 111: the number is 7;
- 0: $len(0) = l - 1$ so the number is long 2 bits;
- 10: the number is 2.

It's decoding time is slow because there is a variable part.

Optimal distribution

Let's recover the optimal distribution for this encoding:

$$|\gamma(x)| = log_2 \frac{1}{p(x)} \implies 2log_2 x - 1 = log_2 \frac{1}{p(x)} \implies$$

$$2log_2 x = log_2 \frac{1}{p(x)} + 1 \implies 2log_2 x = log_2 \left(\frac{1}{p(x)} \cdot 2 \right) \implies$$

$$log_2 x^2 = log_2 \left(\frac{2}{p(x)} \right) \implies x^2 = \frac{2}{p(x)} \implies p(x) = \frac{2}{x^2}$$

10.2.4 δ-code

Given $x \geq 1$:

$$\delta(x) = \gamma(l) bin(x)$$

with $l = |bin(x)|$.

NB: $\delta(l)$ and not $\delta(l-1)$ because we can't encode 0 with δ so no encoding for 1.

Ex: $\delta(9) = 001001001$

NB: $\gamma(9) = 0001001$ is better than $\delta(9)$ but as x increases δ starts to perform better.

Since γ-code was slow in decoding this is even slower.

Optimal distribution

$$|\delta(x)| = |\gamma(l)| + l = 2log_2 l - 1 + l = l + 2log_2 l - 1 = log_2 x + 2log_2 log_2 x - 1$$

$$\implies |\delta(x)| = log_2 \left(\frac{1}{p(x)} \right) \implies p_x = \frac{2}{x}$$

10.2.5 Rice encoding

It's an encoding useful when the values distribution is concentrated nearby a fixed value (like Gaussian distribution). It's a parametric code for k:

$$R_k(x) = U(q)bin(r)$$

with:

- $q = \lfloor \frac{x-1}{2^k} \rfloor$: is the quotient;
- $r = x - 1 - 2^k \cdot q$: is the remainder.

Since $r < 2^k$, because it's a remainder, it stays on k bits. q instead can be 0 too so we use $U(q)$ which stays on $q + 1$ bits.

Ex: $R_4(20) =$:

- $q = \lfloor \frac{20-1}{2^4} \rfloor = \lfloor \frac{19}{16} \rfloor = 1$;
- $r = 20 - 1 - 2^4 \cdot 1 = 19 - 16 = 3$

$R_4(20) = 010011$.

It's decoding is slow since there is a variable length part.

10.2.6 PForDelta

We need codes with faster decoding time, for example codes aligned to byte. PForDelta it's a blocked scheme with parameters b and a base.

Given $S = 5, 4, 6, 7, 9, 15, 9, 20, 4$ we set as *base* the minimum element and shift everything respect to it (called shift to 0) obtaining: $S = 1, 0, 2, 3, 5, 11, 5, 16, 0$. We choose b which is the number of bits used for compression, for example $b = 3$. Then we create two types of elements:

- ESC: an escape sequence, for example 111. 1 sequence;
- all the others. $2^b - 1$ sequences.

Then we compress data: if the number stays on b bits and is different from ESC sequence, we memorize it as it is, otherwise we memorize it as it is in another sequence and put an ESC in the compressed sequence.

Ex: 001 — 000 — 010 — 011 — 101 — 111 — 101 — 111 — 000 is the compressed sequence and the uncompressed sequence must contain just the numbers 11 and 16 with naive encoding (maybe encoding on 8 bits or more).

This decoding is fast because we know the size of the compressed elements so the decoding strategy is immediate.

Rule of thumb

We need to choose b, a common strategy is to choose b so that 90% of S is encoded in b bits.

10.2.7 Elias-Fano code

We take into account increasing sequences: $S = 1, 4, 7, 18, 24, 26, 30, 31$, we define:

- $u = s_n + 1$: in this case 32;
- $b = \lceil log_2 u \rceil$: in this case 5, it should be enough to represent each element;
- $l = \lceil log_2 \frac{u}{n} \rceil$: in this case 2;
- $h = b - l = 3$

we split each number in low and high part:

i	s_i	$bin(b)$	$H(s_i)$	$L(s_i)$
1	1	00001	000	01
2	4	00100	001	00
3	7	00111	001	11
4	18	10010	100	10
5	24	11000	110	00
6	26	11010	110	10
7	30	11110	111	10
8	31	11111	111	11

Table 10.1: Elias-Fano algorithm

we now have high and low parts. We copy low parts one after the other:

$$L = 0100111000101011$$

$$|L| = l \cdot n = n \cdot \lceil log_2 \frac{u}{n} \rceil bits$$

To build the sequence H we count how many high parts there are for each combination and we represent those numbers in negative unary (unary encoding with bits flipped), so:

- 10: because the configuration 000 appears once;
- 110: because the configuration 001 appears twice;
- 0: because the configuration 010 doesn't appear;
- 0: because the configuration 011 doesn't appear;
- 10: because the configuration 100 appears once;
- 0: because the configuration 101 doesn't appear;
- 110: because the configuration 110 appears twice;
- 110: because the configuration 111 appears twice.

$$H = 1011000100110110$$

Some useful notes:

- #1 in $H = n$ because using inverse unary we have a 1 for each actual element in original list;
- #0 in $H = 2^h$ because we have a 0 for each configuration of the higher part.

$$2^h = 2^{b-l} = \frac{2^b}{2^l} = \frac{2^{\lceil log_2 u \rceil}}{2^l} = \lfloor \frac{u}{2^l} \rfloor = \lfloor \frac{u}{2^{\lceil log_2 \frac{u}{n} \rceil}} \rfloor \leq \frac{u}{2^{\lceil log_2 \frac{u}{n} \rceil}} \leq \frac{u}{2^{log_2 \frac{u}{n}}} = \frac{u}{\frac{u}{n}}$$
$$= n$$

so we can say that $|H| = 2n$.

So for a sequence $[u]$ of length n Elias-Fano produces an encoding of:

$$n \left(2 + \lceil log_2 \frac{u}{n} \rceil\right) bits(\text{at most})$$

So we basically have the optimal encoding with an additive term for each symbol (the 2 inside the parenthesis).

Decoding

We can decode linearly because L can be decoded in blocks of l bits and H can be decoded using two counters:

- if we find a 0: we increment the configuration counter;
- if we find a 1: we increment the counter for that configuration and append the next $L(x)$ block.

Random access

The Elias-Fano encoding scheme allows us to random access elements avoiding us to linearly decode everything! To achieve this we first need to define two operations:

- $Rank_1(i)$: returns the count of 1s in the portion $H[1, i]$;
- $Select_1(j)$: returns the position of the j-th bit set to 1.

Ex: $A = 01001110101$ then:

- $Rank_1(6) = 3$
- $Select_1(5) = 9$

NB: we can build a data structure that supports $Select_1(x)$ in $O(1)$!
To answer $Access(i)$:

- low part is basically the i-th group in L: $L[i \cdot l, (i+1) \cdot l]$;
- for the higher part we use $Select_1(i)$ to get the starting position of the right configuration for $H(s_i)$, we need to know the configuration in which this position belongs, so we do $Select_1(i) - i$. With this operation we basically count the 0s up to the position $Select_1(i)$ which says us the correct configuration.

Ex: $Select_1(5) = 11$, so high part is $11 - 5 = 6$, then the high part is 110.
Ex: $Access(3)$:

- $L(s_3) = L[3 * 2, 3 * 3] = 11$;
- $Select_1(3) = 4 \implies 4 - 3 = 1$, then we represent 1 on h bits, so $H(s_3) = 001$.

$Access(3) = 00111$.

Next_GEQ

Another operation we can perform is $Next\_GEQ(x)$ which returns the smallest element in S which is $\geq x$. Es: $Next_GEQ(20) = 24$, $Next_GEQ(18) = 18$.

To be able to perform this operation we need $Select_0(i)$ which returns the position of the i-th 0 in H. Let's for example solve $NextGEQ(25)$:

- we represent 25 in 5 bits: 11001;
- the higher part is 110 which is the 6-th configuration. So since we want something $\geq x$ we need to find the starting point in H of the sixth configuration;
- we use $Select_0(5) + 1 = 11 = p$;
- now we can calculate the number of elements in the configuration before the sixth one using $p - 6 = 5$ and using this we can access L and take the lower parts and linearly scan and compare to get the first one $\geq x$.

Moreover when we get $Select_0(H(x)) + 1$ we can access H and:

- if the bit is 1 we have some members in that bucket, so using L we can get the answer in *log* time because we can perform a binary search in the configuration bucket;
- if the bit is 0 we have no members inside that bucket, so we can directly return the first element of the next bucket, if any.

The actual algorithm is:

```
NextGEQ(x):
    p = select0(H(x)) + 1
    if H[p] == 0:
```

```
        return Access(p-H(x))
    else:
        i = p - H(x)
        jump to the i-th group of l bits in L
        and scan comparing with L(x)
```

Es: $NextGEQ(29)$:

- $H(x) = 111$, $L(x) = 01$
- $p = Select_0(7) + 1 = 14$
- $i = 14 - 7 = 7$
- starting from $L[7 \cdot 2]$ we compare with $L(x) = 01$:
  - we get 10 which is greater than 01 so we return it
- $NextGEQ(29) = H(x)|L(x) = 11101 = 30$

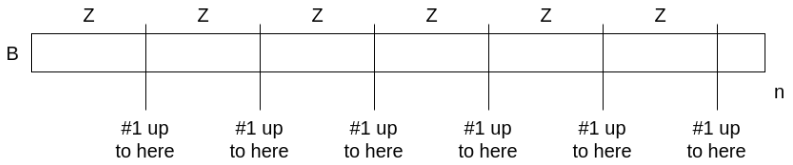
NB: a simplified algorithm (the one we need to know) is:

```
NextGEQ(x):
    p = select0(H(x)) + 1
    i = p - H(x)
    repeat
        y = Access(i)
        i++
    until y >= x
```

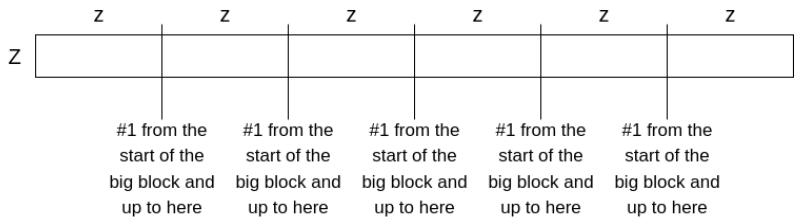
10.2.8 Rank/Select over binary array (dense version)

Suppose we have a binary array $B[1, n]$, we can achieve query time of $O(1)$ and extra space of $o(n)$ bits for rank and select queries.

We start partitioning the array B in blocks of Z bits and associate at each block the number of 1s from the beginning of the B array up to the end of the block:



Then for each of the big blocks we split them into smaller blocks of z bits and associate to each smaller block the number of 1s from the beginning of the bigger block up to the end of the small block:



So to make a query we need to:

- find the big block
- find small block
- scan for the extra 1s

Space occupancy

We store $\frac{n}{Z}$ numbers for the big blocks, each number is $log_2 n$ bits so for the bigger blocks we have $\frac{n}{Z} log_2 n$ bits. Then we need $\frac{n}{z}$ numbers for the smallest blocks each number is $log_2 Z$ bits so the total space occupancy is:

$$\frac{n}{Z} log_2 n + \frac{n}{z} log_2 Z$$

If we want to avoid the scan phase we can pre-compute a table with the data for each small block:

conf. of small block — i	1	2	_	i	_	z
0_0						
0_1						
—						
0010	0	0	1	_	_	_
—						
1_1 (z bits)						

So the table has:

- 2^z rows;
- z cols;
- each cell is $\lfloor log_2 z \rfloor$.

So total space is:

$$\frac{n}{Z} log_2 n + \frac{n}{z} log_2 Z + 2^z \cdot z \cdot log_2 z$$

Common values for Z and z are:

- $Z = log_2^2 n$

- $z = \frac{1}{2} log_2 n$

Let's substitute:

$$\frac{n}{log_2^2 n} log_2 n + \frac{n}{\frac{1}{2} log_2 n} log_2(log_2^2 n) + 2^{\frac{1}{2} log_2 n} \cdot \frac{1}{2} log_2 n \cdot log_2 \left(\frac{1}{2} log_2 n \right) = o(n)$$

NB: since z is 32 we don't use a table but we just read 32 bits and do a pop count with masks.

NB: now we can state that $\exists$ a rank/select data structure that does not touch B, takes $o(|B|) = o(m)$ bits and supports operations in $O(1)$ time, so the total occupancy is:

$$|B| + o(|B|) = m + O(m)$$

Since $n = \#1$ doesn't shows up in the space formula it doesn't depends on n, which means that is useful in the cases in which B is dense!

10.2.9 Rank/Select over binary array (sparse version)

If instead the binary array B is sparse we can use Elias-Fano: we transform B into a sequence of increasing integers and we use the indices of the ones to build our list. Let's call $A = \{2, 3, 8, 10\}$ the array with the positions of the element to 1 in B and $n = 4$ as the size of that array, we can now encode them with Elias-Fano, so:

- $u = 10 + 1 = 11$;
- $b = \lceil log_2 u \rceil = 4$;
- $l = \lceil log_2 \frac{u}{n} \rceil = \lceil log_2 \frac{11}{4} \rceil = 2$;
- $h = b - l = 4 - 2 = 2$.

let's build the table:

x	bin(x)	H(x)	L(x)
2	0010	00	10
3	0011	00	11
8	1000	10	00
10	1010	10	10

and build the sequences:

$$L = 10110010$$
$$H = 11001100$$

So now we can use $NextGEQ$ over the sequences L and H if we can implement select operation over H. To do that we use the other implementation of Rank/Select. So that we:

- $Select_1(x)$ over B is $Access(x)$ over A;
- $Rank_1(x)$ over B is $NextGEQ(x) - 1$ (-1 depends if x is counted in the set or not).

Space occupancy

L is:

$$nlog_2\frac{u}{n} = nlog_2\frac{m}{n}$$

H is:

$$2n$$

In the end the rank/select support over A is $o(n)$, in total:

$$nlog_2\frac{m}{n} + 2n + o(n)$$

and we get:

- select in $O(1)$ time;
- rank in $O\left(log_2\frac{m}{n}\right)$

10.2.10 Variable-byte code (Altavista)

We represent x in binary grouping by 7 bits then we pad them (tag each group) to 8 bit using:

- 0: if this is the last group for this number;
- 1: otherwise

NB: if representation in bit is not a multiple of 7 of course you pad it with some 0s.

Using this encoding we use:

$$8 \cdot \lceil \frac{|bin(x)|}{7} \rceil \text{ bits}$$

Optimal distribution

If we equalize to $log_2\frac{1}{p(x)}$ we get the optimal distribution for this encoding which is:

$$p(x) = \sqrt[7]{\frac{1}{x^8}} = \frac{1}{x^{\frac{8}{7}}}$$

Decoding

To decode we go byte by byte and if the number is greater than 127 (1-st bit is set) we go on with reading and building the number.

10.2.11 (s-c)-dense codes

In the variable-byte code we have two types of configurations:

- *continuers*: configurations of 8 bits that tells us to continue, the one with 1 in front 128, 129, _, 255;
- *stoppers*: configurations of 8 bits that tells us to stop, the ones with 0 in front 0, 1, _, 127.

So with this idea in mind we can create a new class of encoding called *(s-c)-dense codes*:

$$s + c = 2^b$$

with:

- s: the number of stoppers;
- c: the number of continuers;
- b: size of the block in bits.

Ex: for $b = 3$ we can have:

- $s = c = 4$;
- $s = 6, c = 2$.

Of course different combinations of c and s leads to different outcome, for example:

- $s = c = 4$:
  - for single block we can have $s = 4$ different configurations;
  - for two blocks we can have $s \cdot c = 4^2 = 16$ different configurations;
  - for three blocks we can have $s \cdot c^2 = 4^3 = 64$ different configurations;
- $s = 6, c = 2$:
  - for single block we can have $s = 6$ different configurations;
  - for two blocks we can have $s \cdot c = 6 \cdot 2 = 12$ different configurations;
  - for three blocks we can have $s \cdot c^2 = 6 \cdot 2^2 = 24$ different configurations;

In the second configuration if numbers like 4 and 5 are frequent we gain in compression but bits grows faster than the first encoding for larger numbers!

There is an optimal way of partitioning and to choose stoppers and continuers (we won't see those algorithms but are based on dynamic programming). In general making c bigger helps in encoding more numbers in less bits but making s bigger helps with small numbers.

Ex: encode 8 with a (1-3)-dense code: first we need to check if the exercise is correct, so we need to check if $s + c = 2^b \implies 1 + 3 = 4 = 2^2 \implies b = 2$ it is correct. We need to choose the stoppers and the continuers, for example:

- 00: stopper;
- 01: continuer;
- 10: continuer;
- 11: continuer.

Next we tabulate all the configurations up to 8:

- 0: 00;
- 1: 01 00;
- 2: 10 00;
- 3: 11 00;
- 4: 01 01 00;
- 5: 01 10 00;
- 6: 01 11 00;
- 7: 10 01 00;
- 8: 10 10 00;

10.2.12 Interpolative coding

It's an encoding context-aware that exploits patterns in integer sequences, it's very compact as it possibly beats H_0 but it's *very* slow! It use strictly increasing sequences: $S = \{s_1, s_2, \_, s_n\}$ with $s_i < s_{i+1}$. It's also a recursive code.

Let's start with some invariants:

- l: left index;
- r: right index;
- low: a lower bound on the values in $S[l, r]$: $low \leq S[i] \forall i \in S[l, r]$;
- $high$: an upper bound on the values in $S[l, r]$: $high \geq S[i] \forall i \in S[l, r]$.

In encoding phase we start with:

- $l = 1$;
- $r = n$;
- $low = S_1$;
- $high = S_n$.

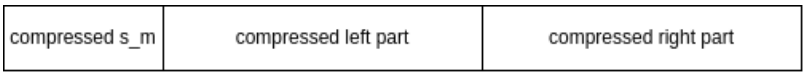
we store those values in the preamble of the compressed stream because it's fundamental that compressor and decompressor starts with the same values!

At each phase with $< l, r, low, high >$ we:

- $m = \lfloor \frac{l+r}{2} \rfloor$;
- we compress s_m given $< l, r, low, high >$ and emit it;
- we recursively compress $< l, m-1, low, s_m - 1 >$;
- we recursively compress $< m+1, r, s_m + 1, high >$.

It's important to notice that in decompression we first decompress s_m, then we can use it to derive the new values for the tuple, so the decompressor knows all the values in the tuple or it can calculate them.

Basically we build a structure like:



Compress single element

Since we know that $low \leq s_m \leq high$ we can say:

$$0 \leq s_m - low \leq high - low$$

We would also like to use a fixed size encoding so we need $b = \lceil log_2(high - low + 1) \rceil$ bits.

Ex: $S = \{1, 2, 5, 6, 8, 9, 10\}$, we pick $l = 1$, $low = 1$, $r = 7$, $high = 10$. Then we calculate $m = 4, s_m = 6$ and we encode $s_m - low = 6 - 1 = 5$ in $b = \lceil log_2(high - low + 1) \rceil = \lceil log_2(10 - 1 + 1) \rceil = 4$ bits. So the final encoded value is 0101.

Using this encoding we are not exploiting all the informations we have! If we use also l, r and the fact that the sequence is strictly increasing we can calculate another bound for s_m:

$$low + (m - l) = low' \leq s_m \leq high' = high - (r - m)$$

so we can now use $b = \lceil log_2(high' - low' + 1) \rceil$ bit.

Ex: $S = 4, 5, 6, 7$ leaving the offset they become $0, 1, 2, 3$ so we store 2 bits!

NB: in some contexts we can emit even 0 bits beacuse if we have, for example, 3 elements in the range 7, 8, 9 we know those items and write no bits.

So during the element decompression we need to check if $r - l = high - low$, if that happens simply emit the elements from $high$ to low.

Exercise

$S = \{1, 2, 3, 5, 7, 9, 11, 15, 18, 19, 20, 21\}$, we pick:

- $l = 1$;
- $r = 12$;
- $low = 1$;
- $high = 21$.

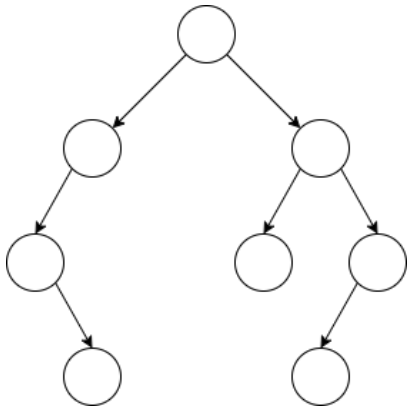
and we start:

- $m = \lfloor \frac{1+12}{2} \rfloor = \lfloor \frac{13}{2} \rfloor = 6$;
- $s_m = 9$;
- $low' = 1 + (6 - 1) = 6$;
- $high' = 21 - (12 - 6) = 15$;
- new range: $6 \leq 9 \leq 15$;
- shifted range: $0 \leq 3 \leq 9$;
- $b = \lceil log_2(15 - 6 + 1) \rceil = 4$;
- we emit 0011; (3 over 4 bits).

The left recursive call uses $< 1, 5, 1, 8 >$ and the right recursive call uses $< 7, 12, 10, 21 >$.

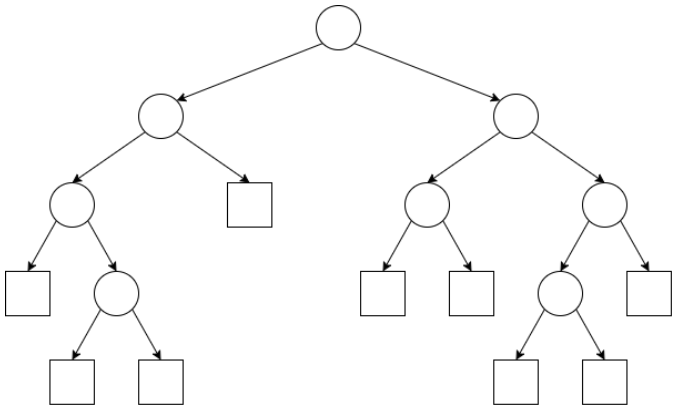
10.2.13 Pointerless programming

Let's implement a binary (search) tree without pointers: let's transform the one in the figure.

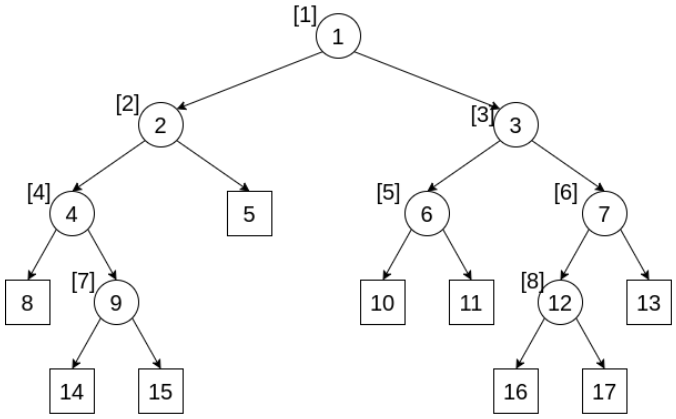


To make it pointerless we need to apply some transformations.

First we *complete the tree*: we fill the tree so that each node is fully binary (each node has two sons):



Then we visit the completed tree in a BFS way and we label each node from 1 to $2n-1$ and also mark the actual real nodes with another incremental counter:



In the end we serialize the structure, to do it we execute another BFS and create a bitvector with 1 in corrispondence of the existing nodes and 0 otherwise:

$$B = 11110110100100000$$

then we build a Rank/Select support over B (using the first implementation), so the space occupancy is:

$$|B| + o(|B|) = 2n + 1 + o(n)$$

So we use 2 bits per node instead of 128 using two pointers for the sons.

Navigate the tree

The labeling is similar to the one we use inside an heap but this is not completely the case since the tree is not actual like heap, in particular we use:

- $left([x]) = 2x$;
- $right([x]) = 2x + 1$.

So the navigation is from the real node number to the completed tree number. To check if the node exists we just need to check in B if the bit in the corresponding plain position is set to 1:

- left child exists iff $B[2x] = 1$;
- right child exists iff $B[2x + 1] = 1$.

To translate from plain number to tagged one we use $Rank(x)$:

- to access to the left child of $[x]$ we access to the index returned by $Rank_1(2x)$;
- to access to the right child of $[x]$ we access to the index returned by $Rank_1(2x + 1)$.

To go up to the parent we do the reverse:

$$parent([x]) = \lceil \frac{Select_1(x)}{2} \rceil$$

Store information

Of course we want to add some information to the nodes of the tree so we store aside from the tree an array in which we use the tagged index from the tree to access the actual element associated to that node.

Exercise

Check if path $\pi[1, p]$ exists starting from the root.

10.3 Data Compression

10.3.1 Statistical compressor

Let's call Σ the alphabet, it can be made from different types of data:

- letters, so we have at most 26 objects;
- words, so we have something $O(millions)$;
- integers, so thery are virtually infinite;

- bytes, so we have 256 objects;
- ...

So depending on the alphabet we change the store and the alphabet size. Since the decompressor must know the alphabet we build a structure before the stream of compressed data, called *preamble* which contains the alphabet representation, the frequencies of the symbols and other informations.

10.3.2 Huffman Encoding

Huffman encoding is a greedy algorithm which gets the optimal encoding for it's class. It builds a tree exploiting the frequencies of the symbols.

Let's assume we have $\Sigma = \{a, b, c, d, e, f\}$ and the frequencies are:

- a: 0.05;
- b: 0.1;
- c: 0.15;
- d: 0.3;
- e: 0.25;
- f: 0.15.

We need to build a tree so we start with just the leaves, one for each symbol:

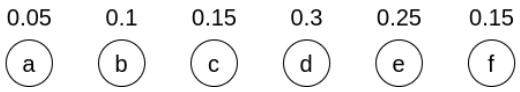


Figure 10.1: Step 0

and we build it joining the two nodes with the lowest frequencies in a single node with frequency equals to the sum of the two single nodes:

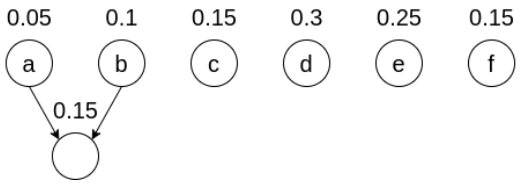


Figure 10.2: Step 1

And we go on in the same way at each round:

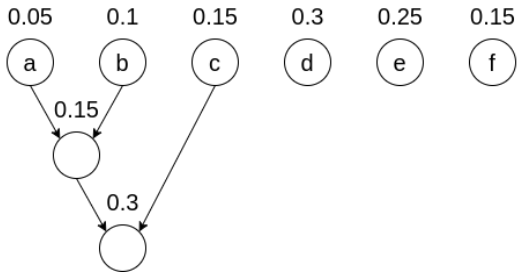


Figure 10.3: Step 2

In the end we get a complete binary tree:

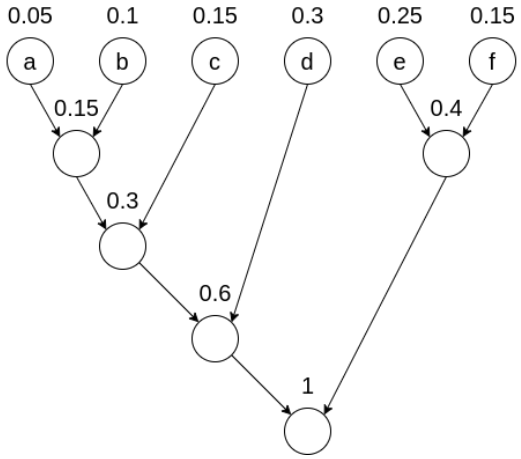
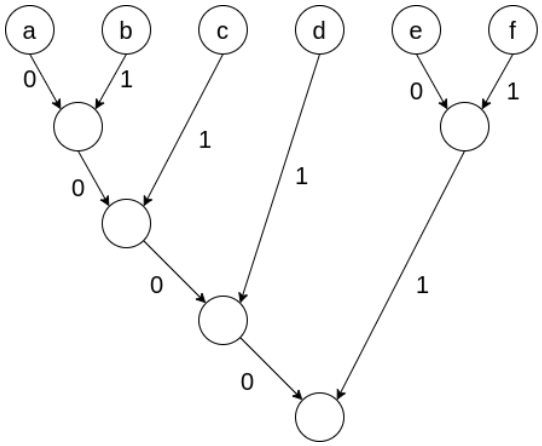


Figure 10.4: Final step

Since it is binary we have $\sigma = |\Sigma|$ leaves, one for each symbol.
To assign an encoding we can:

- assign 0 to each left branch;
- assign 1 to each right branch

and the symbol encoding is just the path to reach it:



So in this case we have the following encoding:

- a: 0000;
- b: 0001;
- c: 001;
- d: 01;
- e: 10;
- f: 11.

It's important to notice that we gain a prefix-free code!

We have $s^{\sigma-1}$ different labeling ways.

NB: whenever we can make different choices to merge nodes it is better to merge the oldest nodes, that minimizes the maximum code-word length. (But the code is always optimal regardless the maximum length!)

To decode a sequence of bits we just need to traverse the tree until we reach a leaf, and that leaf is the encoded character.

Optimality

The average code-word length is:

$$L_c = \sum_{s \in \Sigma} p(s) \cdot |cw(s)|$$

but it is equivalent to the average depth of a leaf.

The key property of this algorithm is that *Huffman code is optimal among all prefix-free codes*, let's prove it.

Lemma 1: let F be the set of binary trees whose average depth is minimum over the binary trees with σ leaves, then $\exists T \in F$ such that the two leaves of smallest $\mathbb{P}$ are at the largest depth and they are children of the same parent:

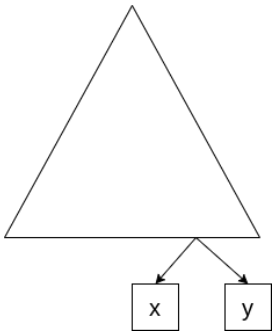
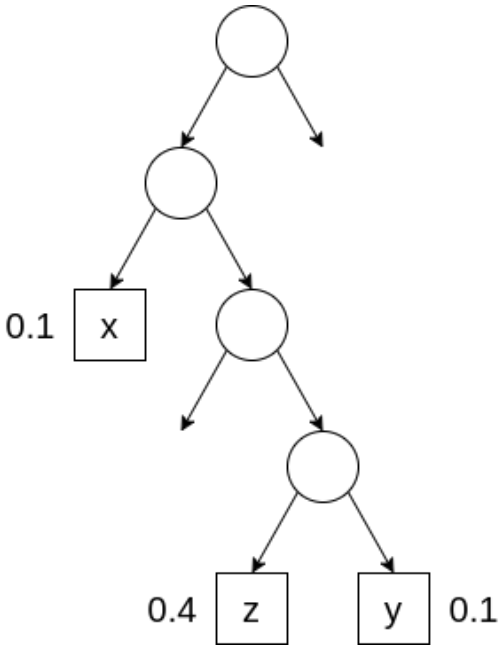


Figure 10.5: x, y have the smallest $\mathbb{P}$

Let's prove the lemma by contraddiction: let's build a tree in which the leaves with the smallest probability are not at the end of the tree:



Let's check the average depth:

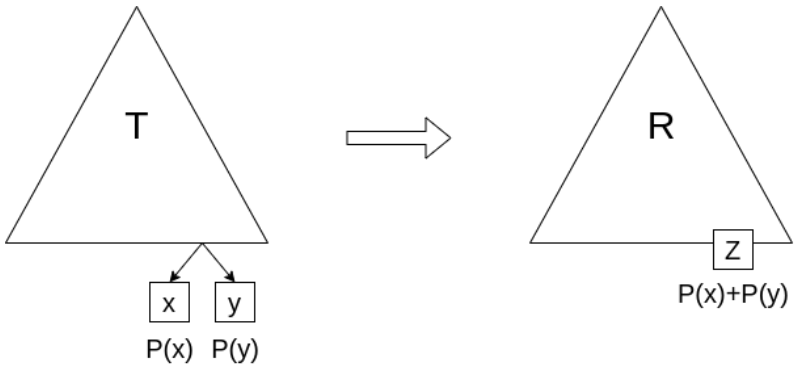
$$0.1 \cdot 3 + 0.4 \cdot 5 + 0.1 \cdot 5 + \_$$

this tree should have the minimum avg depth but if we swap x with z we get:

$$0.4 \cdot 3 + 0.1 \cdot 5 + 0.1 \cdot 5 + \_$$

which for sure is smaller!

Lemma 2: we will study the following reduction in which we join he probability of two leaves into a single node:



We want to know the relation between average depth of both the trees:

$$L_T = \sum_{\text{s leaves}} depth(s) \cdot \mathbb{P}(s)$$

$$= \sum_{\text{s leaves} \neq x,y} depth(s) \cdot \mathbb{P}(s) + depth(x) \cdot \mathbb{P}(x) + depth(y) \cdot \mathbb{P}(y)$$

$$L_R = \sum_{\text{s leaves}} depth(s) \cdot \mathbb{P}(s)$$

$$= \sum_{\text{s leaves} \neq z} depth(s) \cdot \mathbb{P}(s) + depth(z) \cdot (\mathbb{P}(x) + \mathbb{P}(y))$$

so let's study the difference:

$$L_T - L_R = depth(x) \cdot \mathbb{P}(x) + depth(y) \cdot \mathbb{P}(y) - depth(z) \cdot \mathbb{P}(z) =$$

let's call $l = depth(z)$:

$$= (l+1)(\mathbb{P}(x) + \mathbb{P}(y)) - l \cdot (\mathbb{P}(x) + \mathbb{P}(y)) =$$

$$= (l + 1 - l)(\mathbb{P}(x) + \mathbb{P}(y)) =$$

$$= p(x) + p(y)$$

So we can write: $L_T = L_R + \mathbb{P}(x) + \mathbb{P}(y)$

Let's prove that average depth of Huffman tree is minimum: let's call L_H the tree obtained from Huffman encoding algorithm, then we want to prove that $L_H <= L_T \forall T$ with σ leaves. By induction:

- case with $\sigma = 2$: there is only one way to build a tree so it is an Huffman tree and since the only one, it is optimal;

- by induction we impose that Huffman with $\sigma - 1$ symbols is optimal so let's prove that it's true for σ symbols too: by contraddiction let C be an optimal tree that satisfies lemma 1, then L_C is the average depth of C, so we apply the Lemma 2 obtaining Rc, now we can say:

  $$L_C = L_{Rc} + \mathbb{P}(x) + \mathbb{P}(y)$$

  Rc has $\sigma - 1$ leaves s we can apply the inductive hypothesis so there exists an Huffman tree with $\sigma - 1$ leaves that is optimal, let's call it H, now we have:

  $$L_{Rc} \geq L_H$$

  and:

  $$L_C = L_{Rc} + \mathbb{P}(x) + \mathbb{P}(y) \geq L_H + \mathbb{P}(x) + \mathbb{P}(y)$$

  So now we take H which has a leaf with probability $\mathbb{P}(x) + \mathbb{P}(y)$ and split it in two more leaves. The new tree is an Huffman tree because the rest of the tree is an Huffman tree and $\mathbb{P}(x)$ and $\mathbb{P}(y)$ are least probable leaves, let's call it E_H (enlarged), for Lemma 2 we have:

  $$L_{Eh} = L_H + \mathbb{P}(x) + \mathbb{P}(y) \leq L_C$$

  so the Huffman tree obtained, with σ leaves, is optimal!

Performance

It can be proved that:

$$H \leq L_{Huffman} < H + 1$$

in which:

- H: is the entropy of the source;
- $L_{Huffman}$: is the average code-word length in bits.

But of course the problem is $H : 0 \leq H \leq log_2\sigma$, so:

- if $H = 20$, 1 bit means nothing;
- if $H = 2$, 1 bit is a lot.

So it is optimal for random sources but awful for repetitive ones.

One note

$$L_H = \sum_{s \in \Sigma} \mathbb{P}(s) \cdot |cw(s)|$$

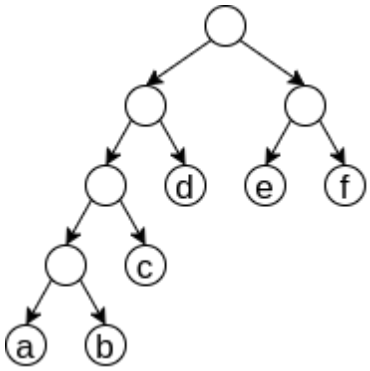
with $|cw(s)| \in \mathbb{N}$,

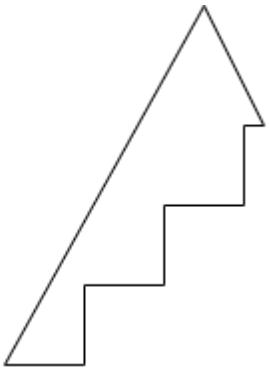
$$H = \sum_{s \in \Sigma} \mathbb{P}(s) \cdot log_2 \frac{1}{\mathbb{P}(s)}$$

with $log_2 \frac{1}{\mathbb{P}(s)} \in \mathbb{R}$. So it could be $H < L_H$ but still be optimal! To avoid fractional bits we can pack many symbols together to spread the additional bit among more symbols!

10.3.3 Canonical Huffman

An Huffman tree in canonical form is a tree formed in the following way:





and it can be stored without the use of any pointer thus speeding up a lot the decompression!

It is faster also because we can encode the first code-word of each level and then derive the other ones from the first one, so we also make the preamble smaller.

Let's call them *first code-word* (f_c):

- a: $0000 = f_c[4]$;
- c: $001 = f_c[3]$;
- d: $01 = f_c[2]$.

So given a code-word the next ones can be obtained just increasing. Of course we need the size of the level bu we can store just the f_c array and a symbol table:

- $f_c[4] = 0000$;
- $f_c[3] = 001$;
- $f_c[2] = 01$;
- $f_c[1] = ?$.

index	symbols
1	//
2	d e f
3	c
4	a b

Using the canonical tree we can check if a bit combination exists on a certain level just checking if we are on the left or on the right of the first code-word of that level.

For the levels with no code-words we store in f_c an element which can't be on that level, greater than everyone else, for example $f_c[1] = 10$ which is surely bigger than 1 and 0 which are the only code-words that can be used on level 1.

The pseudo-code to decode a symbol is:

```
DecodeSymbol()
    v = next_bit()
    l = 1

    while (v < fc[l])
        v = 2*c + next_bit()
        l++
    return symbol[l][v-fc[l]]
```

NB: we could have a cache miss in symbol table access but if symbols aren't too much we can fit entire table in cache.

Build canonical form

To build a canonical Huffman tree we:

- construct the Huffman tree;
- compute the array $num[1, l]$ s.t. $num[i] = \#$symbols at level i in Huffman tree;
- compute symbol table;
- compute first code-word array s.t.:

$$\begin{cases} f_c[l] = 0 \\ f_c[i] = \frac{f_c[i+1]+num[i+1]}{2} \end{cases} \tag{10.1}$$

Once the tree is built we build metadata and then dump the tree.

Encoding algorithm

To encode a symbol we:

- search for the symbol in the symbol table, get l and $offset$;
- the encoding is $f_c[l] + offset$ in l bits.

10.3.4 Arithmetic encoding

It's and encoding which exploits the floating point representation of numbers to map a sequence to a number in the range $[0, 1)$. Of course it is a statistical method so we need to know the frequencies/probabilities for each symbol.

Floating point numbers

Given $x \in [0, 1)$ we can represent it with:

$$x = 0.b_1 b_2 b_3 \_ b_k \_$$

It is a positional representation because each bit represent $\frac{1}{2^k}$ and $b_i \in \{0, 1\}$.

For example: $0.101_2 = \frac{1}{2} + \frac{1}{8} = \frac{5}{8}$. The number in the form:

$$\frac{v}{2^a}$$

with:

- v: the representation of the fractional part of the numbers;
- a: the number of bits of the fractional part.

are called *dyadic fractions*. Ex:

$$0.1011_2 = \frac{11}{16}$$

and $1011_2 = 11$ on 4 digits $2^4 = 16$.

We can make a function to convert any number in this binary representation:

```
Converter(x, k)
    for i in 0..k:
        if 2x < 1:
            emit 0
            x = 2x
        else
            emit 1
            x = 2x - 1
```

This algorithms create the number from left to right so we can stop whenever we want.

Given $x \in [0, 1)$ if we truncate x to the first d bits we get $\tilde{x}$ and we can state that:

$$0 \leq x - \tilde{x} \leq 2^{-d}$$

We can visually prove it:

$$\begin{array}{rccccc|ccc} x & .b_1 & b_2 & b_3 & \_ & b_d & b_{d+1} & b_{d+2} & \_ \\ \tilde{x} & .b_1 & b_2 & b_3 & \_ & b_d & 0 & 0 & \_ \\ x - \tilde{x} & 0 & 0 & 0 & \_ & 0 & b_{d+1} & b_{d+2} & \_ \end{array}$$

so for sure the result has d leading zeroes:

$$x - \tilde{x} = \sum_{i=d+1}^{\infty} b_i \cdot 2^{-i} = \sum_{i=1}^{\infty} b_{d+1} \cdot 2^{-(d+i)} \leq \sum_{i=1}^{\infty} 2^{-(d+i)} =$$

$$= 2^{-d} \sum_{i=1}^{\infty} 2^{-i} = 2^{-d}$$

Encoding

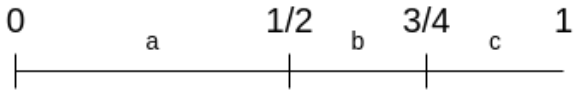
Starting from the frequencies of each symbol:

- $\mathbb{P}(a) = \frac{1}{2}$;
- $\mathbb{P}(b) = \frac{1}{4}$;
- $\mathbb{P}(c) = \frac{1}{4}$.

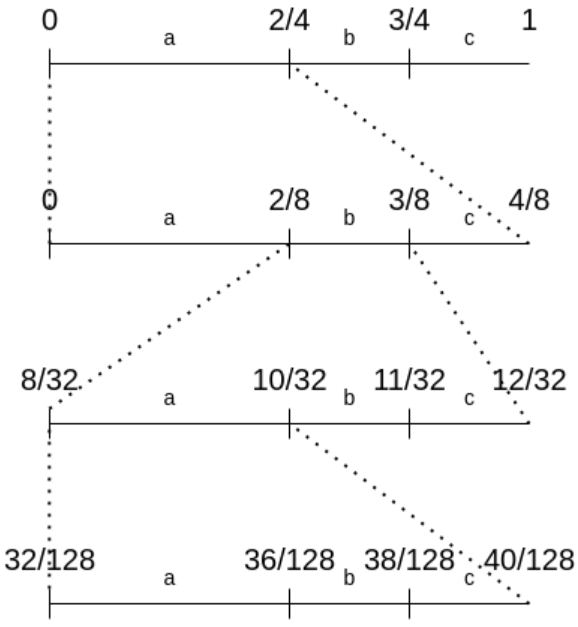
we compute the cumulative frequency:

- $f_a = 0$;
- $f_b = p(a) = \frac{1}{2}$;
- $f_c = p(a) + p(b) = \frac{3}{4}$.

Then starting with the interval $[0, 1)$ we split it according to the probability of each symbol:



Then looping over each symbol of the sequence we focus on the interval specified by the symbol, for example $S = a, b, a, c$:



Formally we use two numbers to remember the range and navigate it:

- l: left interval;
- s: size of the interval.

we start with:

- $l = 0$;
- $s = 1$

and at each iteration we update the values according to:

- $l_i = l_{i-1} + s_{i-1} \cdot f(T[i])$
- $s_i = s_{i-1} \cdot \mathbb{P}(T[i])$

Once we are over with the navigation we compute:

$$x = l + \frac{s}{2}$$

so basically we compute the middle element inside the range. We don't actually need exactly the element in the middle but any number which lays in the range so we can truncate the representation to the minimum number inside the range: $\tilde{x}$ = truncation of x at d bits with:

$$d = \lceil log_2 \frac{2}{s} \rceil \leq 2^{-d}$$

Let's prove that $\tilde{x}$ is in the range:

$$2^{-d} = 2^{-\lceil log_2 \frac{2}{s} \rceil} = \frac{1}{2^{\lceil log_2 \frac{2}{s} \rceil}} \leq \frac{1}{2^{log_2 \frac{2}{s}}} = \frac{1}{\frac{2}{s}} = \frac{s}{2}$$

The size changes according to the probability of the symbols:

- $s_0 = 1$;
- $s_1 = \mathbb{P}(a)$;
- $s_2 = s_1 \cdot \mathbb{P}(b) = \mathbb{P}(a) \cdot \mathbb{P}(b)$;
- $s_2 = s_2 \cdot \mathbb{P}(a) = \mathbb{P}(a) \cdot \mathbb{P}(b) \cdot \mathbb{P}(a)$;

Decoding

We interpret the resulting sequence as a floating point number and check where it lays in those partitions, once we get the partition we get the letter and move on to the next partitioning.

NB: we can compress how many symbols we want so during the decompression we need to know how many symbols we need to decompress!

Performance

$$d = \lceil log_2 \frac{2}{s} \rceil = \lceil 1 + log_2 \frac{1}{s} \rceil = \lceil 1 - log_2 s \rceil < 2 - log_2 s$$

since: $s = \mathbb{P}(a) \cdot \mathbb{P}(b) \cdot \mathbb{P}(a) \cdot \mathbb{P}(c)$ we can say:

$$d < 2 - log_2 \prod_{i=1}^{n} \mathbb{P}(T[i]) = 2 - \sum_{i-1}^{n} log_2 \mathbb{P}(T[i])$$

NB:

$$\sum_{i-1}^{n} log_2 \mathbb{P}(T[i]) = log_2 \mathbb{P}(a) + log_2 \mathbb{P}(b) + log_2 \mathbb{P}(a) + log_2 \mathbb{P}(c) =$$

$$= 2log_2 \mathbb{P}(a) + log_2 \mathbb{P}(b) + log_2 \mathbb{P}(c) =$$

so we have the frequency of a symbol times the logarithm of the probability of the symbol, so:

$$= 2 - \sum_{\sigma} occ(\sigma) \cdot log_2 \mathbb{P}(\sigma) = 2 + \sum_{\sigma} occ(\sigma) \cdot log_2 \frac{1}{\mathbb{P}(\sigma)} =$$

$$= 2 + n \cdot \sum_{\sigma} \frac{occ(\sigma)}{n} \cdot log_2 \frac{1}{\mathbb{P}(\sigma)} = 2 + n \cdot \sum_{\sigma} \mathbb{P}(\sigma) \cdot log_2 \frac{1}{\mathbb{P}(\sigma)}$$

$$= 2 + nH$$

So the total number emitted for all the text is $2 + nH$, then the average bits emitted per symbol is:

$$\frac{d}{n} \leq \frac{nH + 2}{n} = H + \frac{2}{n}$$

The longer the text, the more we are converging to the entropy! But we are assuming infinite size floating point number which is not the case in practice. In the application we get something like $H + \frac{1}{100}$ on fixed size text blocks.

10.3.5 Dictionary-based compressors

Up to now we have seen statistical compressor/coders, which uses frequencies. Now we'll see some dictionary-based compressors which uses syntactic. In particular we'll see the ones from Lempel and Ziv: LZ77 and LZ78 which are the bases of *gzip*, *Brotli*, *ZSTD* and *LZFSE*.

10.3.6 LZ77

Let's consider the followind sequence:

$$T = aacaacabcaaaaaaac$$

and suppose that you've already processed up to *aacaacab*, so we want to encode the next part, we can *copy* the next part *caa* from before and append an *extra* character *a*. So we need to build a dictionary of all substrings starting before the current point. Of course we can't store all the dictionary D.

To compress we search for the biggest prefix string for the string we want to compress, we call that part *copy* and the new character *extra*. So for each sequence we compress we emit a tuple of the type:

$$< distance, length, extra >$$

with:

- distance: the distance from the current mark to the start of the copy section;
- length: size of the copy section;
- extra: extra character to add once copied.

So in that case we would emit: $< 6, 3, a >$.

NB: suppose we need to compress the sequence *aaaaac* and we've already compressed the first character with $< 0, 0, a >$ we can compress the following part with just $< 1, 4, c >$. Since we copy using the following algorithm:

```
for(i=0; i < l; i++)
    T[cursor+i] = T[cursor-d+i]
```

we can also exploit overlapping sequences. An overlap can be detected checking if $l > d$.

NB: this compression in the worst case can grow in size the input data so it is important to compress on a buffer with 1.2 times the initial size.

LZSS

An improved version of LZ77 is LZSS. At the start of LZ77 we copy nothing so the emitted symbol is $< 0, 0, a >$, we can exploit it and emitting two types of symbols:

- $< 0, d >$: if the first element is 0 we don't copy and add a character;
- $< d, l >$: if the first element is not 0 we copy using d and l just like before.

Full example

$$T = aacaacabcaaaaaaac$$

Using LZ77:

- a emits: $< 0, 0, a >$;
- ac emits: $< 1, 1, c >$;

- *aacab* emits: $< 3, 4, b >$;
- *caaa* emits $< 6, 3, a >$;
- *aaaac* emits: $< 1, 4, c >$.

To store the tuples we can build some lists:

$$T_d = \{0, 1, 3, 6, 1\} T_l = \{0, 1, 4, 3, 4\} T_c = \{a, c, b, a, c\}$$

and then we compress each list using Huffman exploiting the even distribution of the symbols.

Using LZSS:

- *a* emits: $< 0, a >$;
- *ac* emits: $< 1, 1 >$;
- *aaca* emits: $< 3, 4 >$;
- *b* emits: $< 0, b >$;
- *caa* emits: $< 6, 3 >$;
- *aaaaa* emits: $< 2, 5 >$;
- *c* emits: $< 8, 1 >$;

Speed up the algorithm

To speed up the algorithm we introduce a *window* which is the maximum distance we can check for whenever we scan back. In *gzip* we can specify as a parameter: -1, -2, _, -9 which means the window size in kilobytes.

To find prefixes faster we can use an hash-table of triples of chars:

- $< aac, 1 >$;
- $< aca, 2 >$;
- $< \_, \_ >$

so if the window is of size w we will have w items in the hash-table.

To compute a match we take a triple, search in the hash-table, we take all the matches and we brute-force the longest match among the ones we have. If there is no match we advance by one character.

We can update the hash-table deleting the triples outside of the window while we advance it and just pushing the new ones. To do it faster we can add pointers to hash-table entries to link each triple to the next one in order, so with just an head and a tail we can delete elements without searching in the hash-table.

10.3.7 LZ78

This time we build up a real dictionary D:

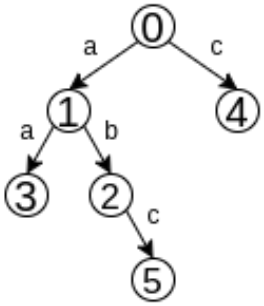
$$T = aabaacabcabcb$$

Initially the dictionary is empty so processing a and searching for the longest copy in D we get the empty string, so we emit the tuple $< 0, a >$ in which the first number is the number of the entry in the dictionary and the character is the *extra* to append. Then we update the dictionary.

Then we go for a, there is a match inside the dictionary so we go for ab, no match so we emit $< 1, b >$ and add this entry to the dictionary.

Then we go for a, there is a match, so we go for aa, no match so we emit $< 1, a >$ and add this entry to the dictionary.

We build the dictionary using a trie since we are looking for prefixes, in the end we will build something like:



Of course the trie grows a lot so at certain point we can freeze the dictionary or discard it and start again from scratch.

Decode

To decode we just need to build the trie again and decode each tuple. To find the node faster in the trie we can keep an array of pointers to the nodes and traverse the tree upward.

10.3.8 Some stats

Compression ratio

Compression ratio is defined as:

$$\frac{\text{size compressed}}{\text{original size}}$$

so the smaller the better.

Compression speed

$$\frac{\text{original size}}{\text{time to compress}}$$

Decompression speed

$$\frac{\text{original size}}{\text{time to decompress}}$$

10.3.9 Burrows-Wheeler transform

This algorithm is the base for the bzip2 command on linux. It's called block sorting compressor and in general it splits data in blocks, each block then is compressed (applying some transformations) and we do some sort inside the block. But we need some tools before.

Move To Front (MTF)

It's a transformation from symbols to symbols that tries to exploit local homogeneity. Let's use the sequence $T = abaabacdccddd$, we transform it to a sequence of small integers:

- we take the alphabet $\Sigma = \{a, b, c, d\}$;
- we start creating new sequence S^{MTF} using list $l = (a, b, c, d)$. Substituting each symbol in T with its position in l, then we move that position to the front of l.

Ex:

- emit 0, push a in front, $l = (a, b, c, d)$;
- emit 1, push b in front, $l = (b, a, c, d)$;
- emit 1, push a in front, $l = (a, b, c, d)$;
- emit 0, push a in front, $l = (a, b, c, d)$;
- emit 1, push b in front, $l = (b, a, c, d)$;
- emit 1, push a in front, $l = (a, b, c, d)$;
- emit 2, push c in front, $l = (c, a, b, d)$;
- emit 3, push d in front, $l = (d, c, a, b)$;
- emit 1, push c in front, $l = (c, d, a, b)$;
- emit 0, push c in front, $l = (c, d, a, b)$;
- emit 1, push d in front, $l = (d, c, a, b)$;

- emit 0, push d in front, $l = (d, c, a, b)$;
- emit 0, push d in front, $l = (d, c, a, b)$;

Using this MTF we can exploit local homogeneity since we use same symbols and sometimes we change context (for example we have 2-3 when we switch from ab to cd).

NB: let's suppose we have this string:

$$X = 1^n 2^n 3^n \_ n^n |X| = n^2$$

for n different numbers. Using Huffman we would have:

$$\mathbb{P}(1) = \mathbb{P}(2) = \_ = \mathbb{P}(n) = \frac{1}{n}$$

so $|cw(1)| = |cw(2)| = \_ = |cw(n)| = log_2 n$ bits, so we would produce $|X| log_2 n = n^2 log_2 n$ bits. Suppose now to pass the sequence X via MTF and encode the result with γ-code:

$$MTF(x) = ?000\_0?000\_0\_?000\_0$$

each block would be formed by:

- one number which indicates the actual position of the new character in l;
- $n - 1$ zeroes because now the element is on the first position of l.

The γ-encode (note that γ-code start from 1, so we shift each number, so we encode $x + 1$) is:

$$\gamma(MTF(X)) = ?111\_1?111\_1\_?111\_1$$

in which each block is formed by:

- a starting which is at most $|\gamma(n)| \leq O(log_2 n)$ bits;
- $n - 1$ bits to 1 to encode the zeroes.

So the whole sequence is at most:

$$O(n \cdot log_2 n) + n(n - 1)$$

so we have $O(n^2)$ bits instead of the $O(n^2 log_2 n)$ bits with Huffman. That's because Huffman is a prefix-free code in which we assign the same code to each symbol. But in the new encoding we don't use prefix-free codes!

Run-Length Encoding (RLE)

Let's suppose here too to have the sequence:

$$a^n b^n c^n \_$$

and we encode it with the tuples: $< a, n >, < b, n >, < c, n >, \_$. For each run of equal symbols we emit a couple. It is good when local homogeneity property is strong!

NB: It was used in FAX machines: suppose we have two lines:

$$01101000111010000000110$$

we xor them obtaining 00101000001 and we encode it with $0, 2, 1, 1, 1, 5, 1$ in which the first symbol says the starting bit, then each number says the size of each run, then it is encoded with Huffman.

Bzip

The algorithm is the following:

- rotate the text and make all the rotating permutations;
- sort the strings;
- take the last row and call it L (we take the last character of each permutation after sorting them);
- move to front code the string L;
- run-length the resulting sequence;
- apply statistical coder.

To come back from L to the original string we need some observations:

- we can compute the first column (F) of the matrix just by sorting L;
- each row of the matrix is a suffix of the original string;
- having F and L we can compute couples of near characters;
- all the occurrences of a same symbol c in L maintain the same order as in F.

Using all the properties described above (in particular the last one) we can rebuild the original string backward starting from the end symbol \$ which will be the the first row.

Example with the string *mississipi*\$:

- let's build the matrix with all the permutations:

```
mississipi$
ississipi$m
ssissipi$mi
sissipi$mis
issipi$miss
```

```
ssipi$missi
sipi$missis
ipi$mississ
pi$mississi
i$mississip
$mississipi
```

- we sort them:

```
$mississipi
ississipi$m
issipi$miss
ipi$mississ
i$mississip
mississipi$
pi$mississi
ssissipi$mi
sissipi$mis
ssipi$missi
sipi$missis
```

- take $L = imssp\$iisis$.

To decode we start with $L = imssp\$iisis$ and:

- we compute F by sorting L, so: $F = \$iiiimpssss$;
- we build the string backwards starting from \$:
  - character before \$ is i;
  - last appearance of i in F is on row 5, so the character before is p;
  - last appearance of p in F is on row 7, so the character before is i;
  - third appearance of i in F is on row 4, so the character before is s;
  - last appearance of s in F is on row 11, so the character before is s;
  - third appearance of s in F is on row 10, so the character before is i;
  - second appearance of i in F is on row 3, so the character before is s;
  - second appearance of s in F is on row 9, so the character before is s;
  - first appearance of s in F is on row 8, so the character before is i;
  - first appearance of i in F is on row 2, so the character before is m

so the recovered string is $mississipi\$$.

Full example

Given $L = abrbb\$aa$, we encode it in $< abrbbaa, 5 >$ in which the second number says the position of the end of string. We apply MTF:

- $l = (a, b, r)$, $a- > 0$;
- $l = (a, b, r)$, $b- > 1$;
- $l = (b, a, r)$, $r- > 2$;
- $l = (r, b, a)$, $b- > 1$;
- $l = (b, r, a)$, $b- > 0$;
- $l = (b, r, a)$, $a- > 2$;
- $l = (a, b, r)$, $a- > 0$;

so $MTF(X) =<< 0, 1, 2, 1, 0, 2, 0 >, < a, b, r >, 5 >$

The RLE is done only on runs of zeroes: for example $X = 1, 0, 0, 0, 0, 2, 0, 0, 0, 1, 2, 3, \_$:

- start adding 1 to the numbers which are not 0:

$$X = 2, 0, 0, 0, 0, 3, 0, 0, 0, 2, 3, 4, \_$$

- write $RLE_0 + 1$:

$$X = 2, bin(5), 3, bin(4), 2, 3, 4, \_$$

- encode:

$$2, 101, 3, 100, 2, 3, 4, \_$$

- drop first one of each representation since it's useless:

$$2, 01, 3, 00, 2, 3, 4, \_$$

So at the end each sequence of 1s and 0s should be interpreted as binary since at first step we replaced each 1 with 2, so no confusion!

In the end we use Huffman on the resulting sequence.

NB: $RLE_0 + 1$ and the drop of the first 1 is called *Wheeler Encoding*.

NB: in bzip2 too we can specify a parameter -1, -2, _, -9 to specify the block size in MBs.