Algorithmique

Pierre Aboulker, Paul Jeanmaire, Tatiana Starikovskaya

26 octobre 2023

Table des matières

Première partie

Lecture 1 - 28/09

Table des matières

1 Organisation

Mail Tatiana : `starikovskaya@di.ens.fr` Homeworks are 30% of the final grade, final (theory from lecture) Textbooks :

— *Introduction to Algorithms* - Cormen, Leiserson, Riverst, Stein
— *Algorithms on strings, trees, and sequences* - Gusfield
— *Approximation Algorithms* - Vazirani
— *Parametrized Algorithms* - Cygan, Fomin, Kowalik, Lokshtanov, Marx, Pilipczuk, Pilipczuk, Saurabh

2 Introduction

Algorithm take Inputs and give an output.

Open Problem 1 (Mersenne Prime)**.** *Find a new prime of form* $2^n - 1$

Algorithms do not depend on the language. Algorithms should be simple, fast to write and efficient. Word RAM model : Two Parts : one with a constant number of registers of w bits with direct access, and one with any number of registers, only with indirect access (pointers). Allows for elementary operations : basic arithmetic and bitwise operations on registers, conditionals, goto, copying registers, halt and malloc. To index the memory storing input of size n with n words, we need register length to verify $w \geq \log n$ Algorithms can always be rewritten using only elementary operations. Complexity :

— $Space(n)$ is the maximum number of memory words used for input of size n

— $Time(n)$ is the maximum number of *elementary* operations used for input of size n

Complexity Notations :

— $f \in \mathcal{O}(g)$ if $\exists n_0 \in \mathbb{N}, c \in \mathbb{R}_+,\ f(n) \leq c \cdot g(n),\ \forall n \geq n_0$

— $f \in \Omega(g)$ if $\exists n_0 \in \mathbb{N}, c \in \mathbb{R}_+,\ f(n) \geq c \cdot g(n),\ \forall n \geq n_0$

— $f \in \Theta(g)$ if $\exists n_0 \in \mathbb{N}, c_1, c_2 \in \mathbb{R}_+,\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n),\ \forall n \geq n_0$

3 Data Structures

3.1 Introduction

Way to store elements of a data base that is created to answer frequently asked queries using pre-processing. We care about space used, construction, query and update time. Can be viewed as an algorithm, which analysed on basics. Containers are basic Data Structures, maintaining the following operations :

1. Random Access : given i, access e_i
2. Access first/last element
3. Insert an element anywher
4. Delete any element

3.2 Array

An array is a pre-allocated contiguous memory area of a *fixed* size. It has random access in $\mathcal{O}(1)$, but doesn't allow insertion nor deletion.

Linear Search : given an integer x return 1 if $e_i = x$ else 0.

Algorithme 1 Linear Search in an Array.
Complexity : Time = $\mathcal{O}(n)$ | Space = $\mathcal{O}(n)$

Input$x$

3.3 Doubly Linked List

Memory area that does not have to be contiguous and consists of registers containing a value and two pointers to the previous and next elements. It has random access in $\mathcal{O}(n)$, access/insertion/deletion at head/tail in $\mathcal{O}(1)$.

Algorithme 2 Insertion in a Doubly Linked List
Complexity : $\mathcal{O}(1)$

InputL, x
$x.next \leftarrow L.head$
if $L.head \neq NIL$ **then**
    $L.head.prev \leftarrow x$
end if
$L.head \leftarrow x$
$x.prev = Nil$

3.4 Stack and Queue

Stack : Last-In-First-Out data structure, abstract data structure. Access/insertion/deletion to top in $\mathcal{O}(1)$.

Open Problem 2 (Optimum Stack Generation)**.** *Given a finite alphabet Σ and $X \in \Sigma^n$. Find a shortest sequence of stack operations push, pop, emit that prints out X. You must start and finish with an empty stack. Current best solution is in $\tilde{\mathcal{O}}(n^{2.8603})$.*

Queue : First-In-First-Out abstract data structure. Access to front, back in $\mathcal{O}(1)$, deletion and insertion at front and back in $\mathcal{O}(1)$.

4 Approaches to algorithm design

Solve smalle sub-problems to solve a large one.

4.1 Dynamic Programming

Break the problem into many closely related sub-problems, memorize the result of the sub-problems to avoid repeated computation

Examples :

Algorithme 3 Recursive Fibonacci Numbers
Complexity : Exponential

RFibo(n) :
Input$n$
if $n \leq 1$ **then**
    return n
end if
return RFibo($n-1$) + RFibo($n-2$)

Algorithme 4 Dynamic Programming Fibonacci Numbers
Time = $\mathcal{O}(n)$ | Space = $\mathcal{O}(n)$

Input$n$
$Tab \leftarrow zeros(n)$ ▷ $zeros(n)$ returns a n-array of zeros.
$Tab[0] \leftarrow 0$
$Tab[1] \leftarrow 1$
for $i \leftarrow 2$ to n **do**
    $Tab[i] = Tab[i-1] + Tab[i-2]$
end for
return Tab[n]

Levenshtein Distance between two strings can be computed in $\mathcal{O}(mn)$ instead of exponential time. Based on `https://arxiv.org/pdf/1412.0348.pdf`, this is the best one can do. RNA folding : retrieving the 3D shape of RNA based on their representation as strings. Currently, we know it is possible to find $\mathcal{O}(n^3)$, in $\tilde{\mathcal{O}}(n^{2.8606})$ and if *SETH* is true, there is no $\mathcal{O}(n^{\omega-\varepsilon})$. We know $\omega \in [2, 2.3703)$

Open Problem 3. *Is there a better Complexity for RNA folding ? What is the true value of ω ?*

Knapsack problem : An optimization problem with bruteforce complexity $\mathcal{O}(2^n)$.

Algorithme 5 Knapsack : Dynamic Programming
Time = $\mathcal{O}(nW)$ | Space = $\mathcal{O}(nW)$

Input W, w, v ▷ Capacity, weight and values vectors.
$KP = zeros(n, W)$
for $i \leftarrow 0$ to n **do**
  $KP[i, 0] = 0$
end for
for $w \leftarrow 0$ to W **do**
  $KP[0, w] = 0$
end for
for $i \leftarrow 0$ to n **do**
  for $w \leftarrow 0$ to W **do**
    if $w < w_i$ **then**
      $KP[i, w] \leftarrow KP[i-1, w]$
    else
      $$KP[i, w] = \max \begin{cases} KP[i-1, w] \\ KP[i-1, w-w_i] + v_i \end{cases}.$$
    end if
  end for
end for
return $KP[n, W]$

4.2 Greedy Techniques

Problems solvable with the greedy technique form a subset of those solvable with DP. Problems must have the optimal substrcture property. Principle : choosing the best at the moment.

Example : The Fractional Knapsack Problem
Algorithm : Iteratively select the greatest value-per-weight ratio.

Théorème 4.2.1. *This algorithm returns the best solution, in time $\mathcal{O}(n \log n)$*

By contradiction. Suppose we have $\frac{v_1}{w_1} \geq \ldots \geq \frac{v_n}{w_n}$. Let $ALG = p = (p_1, \ldots, p_n)$ be the output by the algorithm and $OPT = q = (q_1, \ldots, q_n)$ be optimal.
Assume that $OPT \neq ALG$, let i be the smallesst index such $p_i \neq q_i$. There is $p_i > q_i$ by construct. Thus, there exists $j > i$ such that $p_j < q_j$. We set $q' = (q_1^{'}, \ldots, q_n^{'}) = (q_1, \ldots, q_{i-1}, q_i + \varepsilon, q_{i+1}, \ldots, q_j - \varepsilon \frac{w_i}{w_j}, \ldots, q_n)$
$q^{'}$ is a feasible solution : $\sum_{i=1}^{n} q_i^{'} \cdot w_i = \sum_{i=1}^{n} q_i w_i \leq W$
Yet, $\sum_{i=1}^{n} q_i^{'} \cdot v_i > \sum_{i=1}^{n} q_i \cdot v_i$, ce qui contredit la ■

Deuxième partie

TD 1 - 29/09

Table des matières

5 Mathematical Complexity

5.1 Exercice 1

5.1.1 Question 1

Non : prendre $f = 1$ et $g = exp$.

5.2 Question 2

Non, si g = h = f.

5.2.1 Question 3

Non : Si on a $f = n, g = n^2 \in \Omega(f(n)), h = f \in \Theta(f(n))$ alors $g + h \neq \mathcal{O}(f(n))$

5.3 Exercice 2

On rappelle la formule de Stirling :

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

Immédiatement, on en déduit la première relation.

On a par ailleurs la seconde égalité en passant au logarithme, fonction continue en $+\infty$

5.4 Exercice 3

On rappelle les formules suivantes :

$$\begin{cases} (n+a)^b & = n^b(1+\frac{a}{n})^b \\ (1+\frac{a}{n})^b & = 1 + b\frac{a}{n} + o(\frac{a}{n}) \in [1; 1+ba] \end{cases}$$

Immédiatement, on a la relation souhaité.

6 Data Structures

6.1 Exercice 4

Il suffit de diviser l'array en deux sous arrays de taille $n/2$, une array commençant en $i = 0$, une commençant en $j = -1$ et on stocke les deux indices de fin de la pile courant.

6.2 Exercice 5

6.2.1 Question 1

On définit un algorithme de *reverse* de list en temps linéaire en ajoutant tous les éléments dans une pile puis en dépilant dans une liste. On effectue bien $2n = \mathcal{O}(n)$ opérations.
Il suffit alors de comparer les deux listes en temps linéaire.

6.2.2 Question 2

Pour une liste vide, ou d'un seul élément on renvoie True. On reverse en place la première moitié de la liste et on la compare à la seconde et normalement ça marche.

6.3 Exercice 6

On utilise deux piles : On push dans la première, et on pop de la seconde. Lorsque la seconde pile est vide, on pop de la première et on push dans la seconde, ce qui permet bien de former une pile.

6.4 Exercice 7

6.4.1 Question 1

On utilise les arrays standards et lorsqu'on dépasse la capacité, on double le nombre de cases, qu'on initialise à -1, en stockant l'indice du dernier élément de la liste. On a alors toujours une complexité en espace en $\mathcal{O}(n)$ puisqu'on a toujours au plus $2n$ cases.

6.4.2 Question 2

On effectue la suite suivante d'instructions, pour $n \in \mathbb{N}$:

1. On ajoute $2n$ éléments
2. On retire $n + 1$ éléments
3. On ajoute 1 élément
4. On recommence en modifiant n

6.4.3 Question 3

Il suffit alors d'attendre de passer en dessous de la barre de 25% du tableau rempli. On a bien tout de même une complexité en $\mathcal{O}(n)$.

7 Greedy Algorithms

7.1 Exercice 8

7.1.1 Question 1

Algorithme 6 Greedy Algorithm for Scheduling Problem

Input a ▷ Vecteur de tuples correspondant aux activités
$E \leftarrow ListeVide()$
$Sort(a, (fun : x, y \mapsto x[1] \leq y[1]))$ ▷ On trie les activités par date de fin croissante
$s \leftarrow PileVide()$
$Push(a, s)$ ▷ On ajoute une à une les activités de a dans une pile.
while (**do**s)
  $ac \leftarrow Pop(s)$
  if (**then**ac est compatible)
    $Ajouter(E, ac)$
  end if
end while
return E

Correction. On introduit une solution optimale, la plus proche possible de l'algo. ■

7.1.2 Question 2

On prend $T1 = [1, 2]\,, T2 = [3, 4]\,, T3 = [1.5, 2.5]$

7.1.3 Question 3

Bon on fait de la Programmation Dynamique. Relation de récurrence $\forall i DP(i)$ est le max des poids sur $\{T_1, \ldots, T_i\}$

$$\begin{cases} DP(0) & = 0 \\ DP(i+1) & = \max(DP(i), w_{i+1} + DP(p(i+1))) \text{ où } p(i) \text{ est le dernier indice de la dernière tâche compatible} \end{cases}$$

Troisième partie

Lecture 2 - 5/10

Table des matières

8 Divide and Conquer

Divide a problem into smaller ones to solve those, then combine the solutions to get a solution of the bigger problem.
Example : *Merge Sort* : Its complexity verifies $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \mathcal{O}(n)$. From that we will derive that $T(n) = \mathcal{O}(n \log n)$

9 Analysis of Recursive Algorithms

We have recurrences we want to solve. We have three methods :

9.1 Substitution Method

The method :

1. Guess the asymptotic of $T(n)$
2. Show the answer via induction

For *Merge Sort* : we guess $T(n) \leq c \cdot n \log_2 n, \forall n \geq 2$. We choose c that verifies that property until $n = 6$.
Substituting in the recurrence equation :

$$T(n) \leq cn \log_2 \frac{n}{2} + c \log_2 \frac{n}{2} + c\frac{n+2}{2} + a \cdot n = c\dot{n} \log_2 n + a \cdot n + c \cdot \log_2 n - c\frac{n}{2}$$

If we then choose c so that it is bigger than $20a$ we get :

$$T(n) \leq cn \log_2 n + a \cdot n - c \cdot n/20 \leq cn \log_2 n$$

9.2 Recursion-tree Method

1. Simplify the equation :
   - Delete floors and ceils
   - Suppose n is of a good form
2. Draw a tree, rooted with the added term and the recursive calls
3. Start again, and recursively fill the tree

We get a tree of depth $\log_k n$ if n is divided by k in recursive calls. We can now sum the values of the nodes, to get an approximation, and start verifying.

10 Master Theorem

10.1 The Theorem

Théorème 10.1.1 (Master Theorem)**.** *If we have recurrence equation $T(n) = aT(n/b) + f(n)$ where $a \geq 1, b > 1$ are integers, $f(n)$ is asymptotically positive. Let $r = \log_b a$, we have :*

1. *If $f(n) = \mathcal{O}(n^{r-\varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^r)$*
2. *If $f(n) = \Theta(n^r)$ then $T(n) = \Theta(n^r \log n)$*
3. *If $f(n) = \Omega(n^{r+\varepsilon})$ for some $\varepsilon > 0$, and $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n (regularity condition) then $T(n) = \Theta(f(n))$.*

Remarque 10.1.1.1. *The Master Theorem 10.1.1 does not cover all possible cases for $f(n)$. Example : $f(h) = h^r / \log h$*

Remarque 10.1.1.2. *The Master Theorem 10.1.1 is sometimes called THÉORÈME SUR LES RÉCURRENCES DE PARTITION*

11 The Proof

Plan :

— Analyse the recurrence as if T is defined over reals (continuous version)

— Prove the discrete version

11.1 Continuous Master Theorem

Démonstration.

Lemme 11.1.1. *Define $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq \hat{n} \\ aT(n/b) + f(n) & \text{if } n > \hat{n} \end{cases}$ Then*

$$T(n) = \Theta\left(n^r\right) + \sum_{k=0}^{\lceil \log_b(n/\hat{n}) \rceil - 1} a^k f(n/b^k)$$

Démonstration. In the Recursion-Tree, stopped when the argument of T is smaller than $\hat{n}$ which is when depth is $\lceil \log_b(n/\hat{n}) \rceil - 1$, we get :

$$\begin{aligned} T(n) &\leq \sum_{k=0}^{\lceil \log_b(n/\hat{n}) \rceil - 1} a^k f(n/b^k) + \Theta(a^{log_b(n/\hat{n})}) \\ &= \sum_{k=0}^{\lceil \log_b(n/\hat{n}) \rceil - 1} a^k f(n/b^k) + \Theta(a^{\log_b(n)}) \\ &= \sum_{k=0}^{\lceil \log_b(n/\hat{n}) \rceil - 1} a^k f(n/b^k) + \Theta(n^{\log_b(a)}) \end{aligned}$$

■

Back to the proof :

Lemme 11.1.2. *Define $g(n) = \Theta(n^r) + \sum_{k=0}^{q} a^k f(n/b^k)$ Then :*

1. *If $f(n) = \mathcal{O}(n^{r-\varepsilon})$ then $g(n) = \Theta(n^r)$*
2. *If $f(n) = \Theta(n^r)$ then $g(n) = \Theta(n^r \log n)$*
3. *If $f(n) = \Omega(n^{r+\varepsilon})$ and we have the regularity condition then $g = \Theta(f)$*

Démonstration. 1. Case 1 :

$$g(n) \begin{aligned} &= \Theta(n^r) + \sum_{k=0}^{q} a^k f(n/b^k) \\ &= \Theta(n^r) + \mathcal{O}\left(\sum_{k=0}^{q} a^k (n/b^k)^{r-\varepsilon}\right) \end{aligned}$$

However :

$$\begin{aligned} \sum_{k=0}^{q} a^k (n/b^k)^{r-\varepsilon} &= n^{r-\varepsilon} \sum_{k=0}^{q} (ab^\varepsilon / b^r)^k \\ &= n^{r-\varepsilon} \sum_{k=0}^{\lceil \log_b(n/\hat{n}) \rceil - 1} (b^\varepsilon)^k = \mathcal{O}(n^{r-\varepsilon} (n/\hat{n})^{epsilon}) \end{aligned}$$

Thus : $g(n) = \Theta(n^r)$

2. Case 2 : We have :

$$\begin{aligned} g(n) &= \Theta(n^r) + \sum_{k=0}^{q} a^k f(n/b^k) \\ &= \Theta(n^r) + \Theta\left(\sum_{k=0}^{q} a^k (n/b^k)^r\right) \end{aligned}$$

However :

$$\begin{aligned} \sum_{k=0}^{q} a^k (n/b^k)^r &= n^r \sum_{k=0}^{q} (a/b^r)^k \\ &= n^r \sum_{k=0}^{\lceil \log_b(n/\hat{n}) \rceil - 1} 1 = n^r \Theta(\log_b n/\hat{n}) \end{aligned}$$

3. Case 3 : By induction on k : $a^k f(n/b^k) \le c^k f(n)$. Thus :

$$\sum_{k=0}^{q} a^k f(n/b^k) \le \sum_{k=0}^{q} c^k f(n) = f(n) \sum_{k=0}^{q} c^k = \Theta(f(n))$$

■

We thus have proved the continuous Master Theorem. ■

11.2 Discrete Master Theorem

We have now showed the continuous Master Theorem, following William Kuszmaul, Charles E. Leiserson, *Floors and Ceilings in Divide-and-Conquer Recurrences*, Symposium on Simplicity in Algorithms 2021.

Démonstration. See slides below

Why not to follow CLRS textbook?

Floors and Ceilings in Divide-and-Conquer Recurrences*

William Kuszmaul
Charles E. Leiserson

MIT CSAIL
{kuszmaul,cel}@mit.edu

Abstract

The master theorem is a core tool for algorithm analysis. Many applications use the discrete version of the theorem, in which floors and ceilings may appear within the recursion. Several of the known proofs of the discrete master theorem include substantial errors, however, and other known proofs employ sophisticated mathematics. We present an elementary and approachable proof that applies generally to Akra-Bazzi-style recurrences.

include the claim that the theorem holds in the presence of floors and ceilings.

To distinguish the two situations, we call the master theorem without floors and ceilings the ***continuous master theorem***[1] and the master theorem with floors and ceilings the ***discrete master theorem***. When we speak only of the master theorem, we mean the discrete master theorem, but we usually include the term "discrete" in this paper for clarity in distinguishing the two cases.

proved the theorem for exact powers of b. Cormen, Leiserson, and Rivest [5, Section 4.3] presented the discrete master theorem, extending Bentley, Haken, and Saxe's earlier treatment to include floors and ceilings, but their proof is at best a sketch, not a rigorous argument, and it leaves key issues unaddressed. These problems have persisted through two subsequent editions [6, 7] with the additional coauthor Stein.

Why not to follow CLRS textbook?

- Aho, Hopcroft, Ullman offered one of the first treatments of divide-and-conquer recurrences, giving three cases for recurrences of the form T(n) = aT (n/b) + cn (1974)
- Bentley, Haken, and Saxe introduced the master theorem in modern form, but proved it for $n = b^k$ only (1980)
- CLRS extended the proof to the discrete version, but gave only a sketch of the proof (1990)
- Akra and Bazzi considered $T(n) = \sum_{i=1}^{t} a_i T(n/b_i) + f(n)$ (1998)
- Leighton simplifies the proof of Akra and Bazzi and extends is to the discrete version (1996)
- Campbell spots several flows in the proof of Leighton and devotes more than 300 pages to carefully correct the issues (2020)
- More generalizations by Drmota and Szpankowski(2013), Roura (2001), Yap (2011)

Definitions

Discrete recurrences

$$T(n) = f(n) + \sum_{i \in S} a_i T(\lfloor n/b_i \rfloor) + \sum_{i \notin S} a_i T(\lceil n/b_i \rceil)$$

$a_i \in \mathbb{R}^+, b_i \in \mathbb{R}^+, n \geq \hat{n}$

For $1 \leq n < \hat{n}$, there exist c_1, c_2: $c_1 \leq T(n) \leq c_2$

Polynomial-growth condition

$\exists \hat{n} > 0$ such that $\forall \Phi \geq 1 \, \exists d > 1 : d^{-1} f(n) \leq f(\varphi n) \leq d f(n)$

for all $1 \leq \varphi \leq \Phi$ and $n \geq \hat{n}$

6 technical slides ahead!

KEEP CALM AND CARRY ON

Lemma 1. For $\beta > 1, n \in \mathbb{N}$ let $L = \Pi_{i=1}^{n}(1 - \dfrac{1}{\beta^i + 1})^2$, $U = \Pi_{i=1}^{n}(1 + \dfrac{1}{\beta^i - 1})^2$

We have $L = \Omega(1)$ and $U = O(1)$.

Proof.

$$\beta > 1 \Rightarrow \frac{1}{\beta^i} < \frac{1}{\beta^i - 1} \Rightarrow 1/L = \Pi_{1=1}^{n}(1 + \frac{1}{\beta^i})^2 < \Pi_{i=1}^{n}(1 + \frac{1}{\beta^i - 1})^2 = U$$

$$U = \Pi_{i=1}^{n}(1 + \frac{1}{\beta^i - 1})^2 \leq \Pi_{i=1}^{\infty}(1 + \frac{1}{\beta^i - 1})^2 \leq \Pi_{i=1}^{\infty}(e^{1/(\beta^i - 1)})^2 =$$

(Here we use $1 + 1/x \leq e^{1/x}$ for $x \neq 0$)

$$= \exp(\sum_{i=1}^{\infty} \frac{2}{\beta^i - 1}) \leq \exp(\sum_{i=1}^{\infty} \frac{4}{\beta^i}) + O(1) = O(1)$$

Lemma 2. Let $\beta > 1; \beta_i \geq \beta, 1 \leq i \leq k; B := \Pi_{i=1}^{k}\beta_i$

There exists $c = c(\beta) > 0$ such that for all $n_1, n_2, \ldots, n_k$ where $n_i > \max(\beta, 1 + 1/(\sqrt{\beta} - 1))$ and $\lfloor n_{i-1}/\beta_i \rfloor \leq n_i \leq \lceil n_{i-1}/\beta_i \rceil$, we have $c^{-1/4}(n_0/B) \leq n_k \leq c^{1/4}(n_0/B)$.

Proof. Let $\rho_i := \dfrac{n_i}{n_{i-1}/\beta_i}$.

$$(n_0/B)\Pi_{i=1}^{k}\rho_i = \frac{n_0 \Pi_{i=1}^{k}\rho_i}{\Pi_{i=1}^{k}\beta_i} = n_0 \Pi_{i=1}^{k} \frac{\rho_i}{\beta_i} = n_0 \Pi_{i=1}^{k} \frac{n_i}{n_{i-1}} = n_k$$

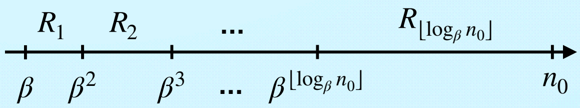
It is enough to show that $c^{-1/4} \leq \Pi_{i=1}^{k}\rho_i \leq c^{1/4}$ for some $c = c(\beta)$

$$n_{i-1}/\beta_i - 1 \leq \lfloor n_{i-1}/\beta_i \rfloor \leq n_i \leq \lceil n_{i-1}/\beta_i \rceil \leq n_{i-1}/\beta_i + 1 \Rightarrow$$

$$n_i - 1 \leq n_{i-1}/\beta_i \leq n_i + 1 \Rightarrow \boxed{\underbrace{\frac{n_i}{n_i + 1}}_{1 - \frac{1}{n_i + 1}} \leq \rho_i \leq \underbrace{\frac{n_i}{n_i - 1}}_{1 + \frac{1}{n_i - 1}} \ (*)}$$

Proof of Lemma 2 (continued).

$$\boxed{\underbrace{\frac{n_i}{n_i+1}}_{1-\frac{1}{n_i+1}} \le \rho_i \le \underbrace{\frac{n_i}{n_i-1}}_{1+\frac{1}{n_i-1}} \;(*)}$$



From $(*)$: $\rho_i \le 1 + \dfrac{1}{n_i - 1} \le 1 + \dfrac{1}{1/(\sqrt{\beta}-1)} = \sqrt{\beta}$

$n_{i+2} = \dfrac{n_i \rho_{i+1}\rho_{i+2}}{\beta_{i+1}\beta_{i+2}} \le n_i/\beta \Rightarrow$ every range R_j contains at most two n_i's

From $(*)$ again: $n_i \in R_j \Rightarrow 1 - \dfrac{1}{\beta^j+1} \le \rho_i \le 1 + \dfrac{1}{\beta^j - 1} (n_i > \beta^j)$

Therefore, $\Pi_{i=1}^k \rho_i = \Pi_{j=1}^{\lfloor \log_\beta n_0 \rfloor} (\Pi_{n_i \in R_j} \rho_i) \le \Pi_{j=1}^{\lfloor \log_\beta n_0 \rfloor} (1 + \dfrac{1}{\beta^j - 1})^2 \le c^{1/4}$ **(Lemma 1)**

$\Pi_{i=1}^k \rho_i \ge \Pi_{j=1}^{\lfloor \log_\beta n_0 \rfloor} (1 - \dfrac{1}{\beta^j + 1})^2 \ge c^{-1/4}$ **(Lemma 1)**

Lemma 3. $\beta_{\min}, \beta_{\max} > 1$. Assume that for all $1 \le i \le k$, $\beta_{\min} \le \beta_i \le \beta_{\max}$, and let $B = \Pi_i \beta_i$.

There exists $c = c(\beta_{\min}, \beta_{\max})$ such that for any $n_1, n_2, \ldots, n_k$ with $n_0 \ge cB$ and $\lfloor n_{i-1}/\beta_i \rfloor \le n_i \le \lceil n_{i-1}/\beta_i \rceil$, we have $c^{-1}(n_0/B) \le n_k \le c(n_0/B)$.

Proof.

Let $c = c(\beta_{\min})$ be the constant from Lemma 3. W.l.o.g. $\sqrt{c} > \max\{\dfrac{1}{\sqrt{\beta_{\min}} - 1} + 1, \beta_{\min}\}$ $(*)$ and $c^{1/4} > 2\beta_i$

If $n_j \ge \sqrt{c}$ for all j, then **Lemma 3** follows from **Lemma 2** and $(*)$. Let j be the smallest value such that $n_j \le \sqrt{c}$. We have $j \ge 1$ as $n_0 \ge cB \ge \sqrt{c}$.

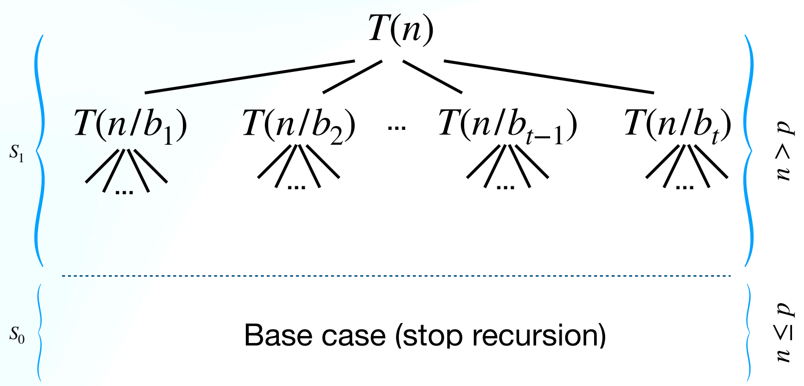
- If $j = 1$, then $n_{j-1} = n_0 \ge c^{-1/4}(n_0/B)$ (trivial).
- If $j > 1$, we apply **Lemma 2** to $\beta_1, \beta_2, \ldots, \beta_{j-1}$ and $n_0, n_1, \ldots, n_{j-1}$ and $\beta = \beta_{\min}$ (all conditions are satisfied) to obtain that $n_{j-1} \ge c^{-1/4}(n_0/B)$

In both cases, $n_{j-1} \ge c^{-1/4}(n_0/B) \ge c^{3/4}$. Therefore, $n_j \ge \lfloor \underbrace{n_{j-1}}_{\ge c^{1/4} > 2\beta_j} /\beta_j \rfloor \ge n_{j-1}/(2\beta_j) > n_{j-1}/c^{1/4} \ge \sqrt{c}$

Lemma 4. Let $a_1, a_2, \ldots, a_t > 0$ and $b_1, b_2, \ldots, b_t > 1$ be constants, $f(n) : \mathbb{R}^+ \to \mathbb{R}^+$ which satisfies the polynomial-growth condition. Consider $T(n) = f(n) + \sum_{i=1}^{t} a_i T(n/b_i)$ defined for $n \in \mathbb{R}^+$ ($*$). Assume that $T'(n)$ defined on $\mathbb{N}$ also satisfies ($*$), but each n/b_i is replaced with $\lfloor n/b_i \rfloor$ or $\lceil n/b_i \rceil$. Then $T'(n) = \Theta(T(n))$.

Proof.

Let c be the constant from Lemma 3 for $\beta_{\min} = \min_i b_i$ and $\beta_{\max} = \max_i b_i$. Let $\hat{n}$ be a sufficiently large constant. Define $p := \max\{\hat{n}, c \cdot \max_i b_i\}$. For $T(n)$, the base case is $n \leq p$.



Subproblem q at level j:
$q = < q(1), q(2), \ldots, q(j) >$
$q(i) \in \{1,2,\ldots,t\}, |q| = j$
$q(i)$ - choice of the recursion brunch at level i
$n_q = n/\Pi_{i=1}^{|q|} b_{q(i)}$

Proof.

$$T(n) = \sum_{q \in S_1} f(n_q)\Pi_{i=1}^{|q|} a_{q(i)} + \sum_{q \in S_0} T(n_q)\Pi_{i=1}^{|q|} a_{q(i)} + f(n) = \sum_{q \in S_1} f(n_q)\Pi_{i=1}^{|q|} a_{q(i)} + \Theta\Big(\sum_{q \in S_0} \Pi_{i=1}^{|q|} a_{q(i)}\Big) + f(n)$$

When computing $T'(n)$ for a subproblem q:

$$\left\lfloor \frac{n'_{<q(1),q(2),\ldots,q(j-1)>}}{q(j)} \right\rfloor \leq n'_q \leq \left\lceil \frac{n'_{<q(1),q(2),\ldots,q(j-1)>}}{q(j)} \right\rceil$$

$$T'(n) = \sum_{q \in S_1} f(n'_q)\Pi_{i=1}^{|q|} a_{q(i)} + \sum_{q \in S_0} T'(n'_q)\Pi_{i=1}^{|q|} a_{q(i)} + f(n) \; (*)$$

As $n_q > p$ for $q \in S_1$, $n_q > p/\max_i b_i \geq c$ for all $q \in S$. By Lemma 3 with $\beta_i = b_{q(i)}$, for all q we have $n'_q = \Theta(n_q)$. It follows that $\exists \Phi > 1$ such that $n'_q \in [\Phi^{-1} n_q, \Phi n_q]$. Therefore, $n'_q \geq n_q/\Phi \geq \hat{n}/\Phi$ and we can choose $\hat{n}$ so that ($*$) is defined correctly.

By the polynomial-growth condition, $f(n'_q) = \Theta(f(n_q))$ for all $q \in S$. For $q \in S_0$, $n'_q = \Theta(1)$ and therefore $T'(n'_q) = \Theta(1)$. It follows:

$$T'(n') = \sum_{q \in S_1} \Theta(f(n_q))\Pi_{i=1}^{|q|} a_{q(i)} + \Theta\Big(\sum_{q \in S_0} \Pi_{i=1}^{|q|} a_{q(i)}\Big) + f(n) = \Theta(T(n))$$

Proof.
$$T(n) = \sum_{q \in S_1} f(n_q)\Pi_{i=1}^{|q|} a_{q(i)} + \sum_{q \in S_0} T(n_q)\Pi_{i=1}^{|q|} a_{q(i)} + f(n) = \sum_{q \in S_1} f(n_q)\Pi_{i=1}^{|q|} a_{q(i)} + \Theta(\sum_{q \in S_0} \Pi_{i=1}^{|q|} a_{q(i)}) + f(n)$$

When computing $T'(n)$ for a subproblem q:
$$\lfloor \frac{n'_{<q(1),q(2),\ldots,q(j-1)>}}{q(j)} \rfloor \le n'_q \le \lceil \frac{n'_{<q(1),q(2),\ldots,q(j-1)>}}{q(j)} \rceil$$

$$T'(n) = \sum_{q \in S_1} f(n'_q)\Pi_{i=1}^{|q|} a_{q(i)} + \sum_{q \in S_0} T'(n'_q)\Pi_{i=1}^{|q|} a_{q(i)} + f(n) \; (*)$$

As $n_q > p$ for $q \in S_1$, $n_q > p/\max_i b_i \ge c$ for all $q \in S$. By Lemma 3 with $\beta_i = b_{q(i)}$, for all q we have $n'_q = \Theta(n_q)$, and hence $\exists \Phi > 1$ such that $n'_q \in [\Phi^{-1} n_q, \Phi n_q]$. Therefore, $n'_q \ge n_q/\Phi \ge \hat{n}/\Phi$ and we can choose $\hat{n}$ so that $(*)$ is defined correctly.

By the polynomial-growth condition, $f(n'_q) = \Theta(f(n_q))$ for all $q \in S$. For $q \in S_0$, $n'_q = \Theta(1)$ and therefore $T'(n'_q) = \Theta(1)$. It follows:

$$T'(n') = \sum_{q \in S_1} \Theta(f(n_q))\Pi_{i=1}^{|q|} a_{q(i)} + \Theta(\sum_{q \in S_0} \Pi_{i=1}^{|q|} a_{q(i)}) + f(n) = \Theta(T(n))$$

Discrete Master theorem

$T(n) = a_1 T(\lfloor n/b \rfloor) + a_2 T(\lceil n/b \rceil) + f(n)$, where
$a := a_1 + a_2 \ge 1$, $b > 1$, $f(n)$ - asymptotically positive.

Define $r := \log_b a$.

Case 1. If $f(n) = O(n^{r-\varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^r)$.

Case 2. If $f(n) = \Theta(n^r)$, then $T(n) = \Theta(n^r \log n)$.

Case 3. If $f(n) = \Omega(n^{r+\varepsilon})$ for some $\varepsilon > 0$, and if $a_1 f(\lfloor n/b \rfloor) + a_2 f(\lceil n/b \rceil) \le c f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Discrete Master theorem

Case 1.

Fact. Replacing $f(n)$ with a function $f'(n)$ satisfying $f'(n) \leq f(n)$ (resp. $f'(n) \geq f(n)$) for all n in the domain of f does not increase (resp. decrease) $T(n)$.

Let $f(n) = O(n^c)$ for $c < \log_b a$. Then as a "bigger" function consider $f'(n) = r(n^c + 1)$ for r big enough. By Lemma 4 and the continuos Master theorem, $T(n) = O(n^{\log_b a})$.

As a "smaller" function, consider $f'(n) = 0$. By Lemma 4 and the continuos Master theorem, $T(n) = \Omega(n^{\log_b a})$.

Exercise. Both bigger and smaller functions satisfy the polynomial growth condition.

Case 2. Analogous.

Discrete Master theorem

Case 3.

$T(n) \geq f(n)$ and hence $T(n) = \Omega(f(n))$. It remains to show that $T(n) = O(f(n))$.

Regularity condition: $a_1 f(\lfloor n/b \rfloor) + a_2 f(\lceil n/b \rceil) \leq cf(n)$ for some $c < 1$ and all $n \geq p$.

For all $n < p$, there exists $s \geq 1$: $T(n) \leq sf(n)$. We show by induction that for all $n \in \mathbb{N}$, $T(n) \leq qf(n)$ for $q = s/(1-c)$.

- Base case: $n < p$ - by the choice of s
- Suppose that $n \geq p$ and the claim holds for all smaller n

$T(n) = a_1 T(\lfloor n/b \rfloor) + a_2 T(\lceil n/b \rceil) + f(n) \leq a_1 q f(\lfloor n/b \rfloor) + a_2 q f(\lceil n/b \rceil) + f(n) \leq$

$$\leq qcf(n) + f(n) = (\frac{sc}{1-c} + 1) f(n) = \frac{s - \overbrace{(1-c)s + 1 - c}^{\leq 0}}{1-c} f(n) \leq qf(n)$$

$\blacksquare$

Remarque 11.2.0.1 (Remarks on the Proof). — *Lemmas 1 to 3 serve to show that the argument does not go too far when it is rounded up or down.*

— *Slide 36 Last Line :* $\frac{2}{\beta^i - 1} < \frac{4}{\beta^i}$ *for* $i \geq i_0$. *Thus :* $\sum_{i=1}^{\infty} \frac{2}{\beta^i - 1} < \sum_{i=1}^{\infty} \frac{4}{\beta^i} + \sum_{i=0}^{i_0} \frac{2}{\beta^i - 1}$ *and that last sum is* $\mathcal{O}(1)$

— *Slide 37 Line 3 : The first inequalities comes from the Recursion-Tree, so that we can ensure the argument does not deviate to much, by the second inequalities.*

11.3 Use Cases

Using the Master Theorem we can show the complexity of many algorithms :

1. Merge Sort Complexity : $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \mathcal{O}(n) = \Theta(n \log n)$
2. Strassen's Algorithm for Matrix Multiplication : $T(n) = 7T(n/2) + \Theta(n^2) \Rightarrow T(n) = \mathcal{O}(n^{\log_2 7}) = \mathcal{O}(n^{2.8074})$

12 Fast Multiplication of Polynomials

The sum of two degree n polynomials can be done in $\mathcal{O}(n)$, Horner's rule for evaluation produces $\mathcal{O}(n)$ complexity. The naïve product can be done in $\mathcal{O}(n^2)$
Remembering Lagrange's Theorem on Polynomials (or Vandermonde's Determinant, or anything really), degree n polynomials are entirely represented by their point-value reprensentation over n distinct points (x_i, y_i). Then, by converting the coefficient reprensentation to point-value representation, then by point-wise multiplicating the polynomials, then by going back to the coefficient representation, we can have a better algorithm.

12.1 Point-Value Multiplication

It is easily done in $\mathcal{O}(n)$ if both polynomials are represented over the same axis.

12.2 Coefficient to Point-Value Conversion - Fast Fourier Transform

For $P = \sum_{i=0}^{n-1} a_i x^i$, we define :

$$\begin{aligned} P_{odd}(x) &= a_{n-1}x^{n/2-1} + a_{n-3}x^{n/2-3} + \ldots + a_1 x \\ P_{even}(x) &= a_{n-2}x^{n/2-2} + a_{n-4}x^{n/2-4} + \ldots + a_2 x^{2/2} + a_0 \end{aligned}$$

1. We have : $P = xP_{odd}(x^2) + P_{even}(x^2)$
2. We evaluate P_{odd}, P_{even} at $(\omega_n^i)^2$ recursively by the halving property.
3. We combine the result.

12.3 Point-Value to Coefficient Conversion - Inverse Fast Fourier Transform

Théorème 12.3.1. $V_n^{-1}[i,j] = \omega_n^{-ij}/n$

Quatrième partie

Lecture 3 - 12/1

Table des matières

Answering queries of appartenance on a set, maintaining a dictionary. Python dictionaries do not have upper bound guarantees, we shouldn't use them.
Suppose we have a set S over a universe U. We dnote by n the number of keys, m the size of the hash table

13 Naïve Array-based implementation

Here we assume that no two objects have equal keys. We store 1 in a bitvector of length $|U|$ at each position i such that i is in S.
This takes $\mathcal{O}(|U|)$ space, for $\mathcal{O}(1)$ search time, and $\mathcal{O}(1)$ modification time.

14 Chained Hash Tables

We give ourselves a function $h : U \to [1, m]$. We send the keys to their image by h in a table. However, since we have no guarantee that $|U| \leq m$, there might be collisions.
To deal with that, instead of storing in the table a boolean, we store a list of all the keys corresponding to $h(k)$.
Then we insert a key in constant time $\mathcal{O}(1)$, search a key in time $\mathcal{O}(|h[key]|)$ and delete a key in time $\mathcal{O}(|h[key]|)$ since in the worst case the key is at the end of the list.

Further analysis leads to the following theorem.

Théorème 14.0.1 (Simple Uniform Hashing Assumption)**.** *Assuming SUHA : « h equally distributes the keys into the table slots », and assuming $h(x)$ can be computed in $\mathcal{O}(1)$, $E\left[T_{search}(n)\right] =$*

$\mathcal{O}\left(1+\frac{n}{m}\right)$, *and same for deletion time.*
Formally, SUHA is :

$$\forall y \in [1, |T|] \mathbb{P}\left(h(x) = y\right) = \frac{1}{|T|}$$

$$\forall y_1, y_2 \in [1, |T|]^2 \mathbb{P}\left(h(x_1) = y_1,\ h(x_2) = y_2\right) = \frac{1}{|t|^2}$$

Démonstration. 1. Unsuccessful Search : Suppose that $k_0, \dots, k_{n-1}$ are keys in the dictionary and we perform an unsuccessful search for a key k. The number of comparisons is : $\sum_{i=0}^{n-1} \mathbb{1}_{h(k)=h(k_i)}$. Then, the expected time is : $\mathbb{E}[T_{search}] = \frac{n}{|T|}$ by SUHA.

2. Successful Search : Suppose keys were introduced in order $k_0, \dots, k_{n-1}$. k_i appears before any of $k_0, \dots, k_{i-1}$ and after any of $k_{i+1}, \dots, k_{n-1}$ that are in the same linked list. Then, to search for k_i, we need $\sum_{j=i+1}^{n-1} \mathbb{1}_{h(k_j)=h(k_i)}$. Under SUHA, the expectation of each of these variables is $\frac{1}{|T|}$. Then, the average expected search time is : $\frac{1}{n}\sum_{i=0}^{n-1} = 1 + \frac{1}{n}\sum_{i=0}^{n-1} \frac{(n-1-i)}{|T|} = \mathcal{O}(1 + \frac{n}{|T|})$

This concludes the proof of the theorem. ■

Good hash functions are functions that distribute the keys evenly. Yet, we do not know what the keys are, and thus will need various heuristics to answer this question. At least, without loss of generality, we can assume that keys are integers.

14.1 Heuristic Hash Functions

- Division Method : $h(k) = k \mod m$. It is better to choose m to be a prime number, and avoid $m = 2^p$.
- Multiplication Method : $h(k) = \lfloor m \{k \cdot A\} \rfloor$. $A \in (0, 1)$ and $m = 2^p$.

Yet, fixing the function can allow anyone to construct a probability distribution for which the function will be "bad".

14.2 Universal Family of Hash Functions

$H = \{h : U \to [0, |T| - 1]\}$ is Universal if :

$$\forall k_1 \neq k_2 \in U,\ |\{h \in H \mid h(k_1) = h(k_2)\}| \leq \frac{|H|}{m}$$

Théorème 14.2.1. *If h is a hash function chosen uniformly at random from a universal family of hash functions. Suppose that $h(k)$ can be computed in constant time and there are at most n keys. Then the expected search time is $\mathcal{O}(1 + \frac{n}{|T|})$*

Démonstration. Same as the case when h satisfies SUHA. Observe that the probability comes this time from choosing the function. ■

Théorème 14.2.2. *Let $p \in \mathcal{P}$ such that $U \subseteq [0, p-1]$. Then $H = \{h_{a,b}(k) = ((ak + b) \mod p) \mod |T| \mid a \in \mathbb{Z}_p^*,\ b$ is a universal family.*

Démonstration. Let $k_1 \neq k_2 \in U$. Let $l_i = (ak_i + b) \mod p$. We have $l_1 \neq l_2$ and $a = ((l_1 - l_2)((k_1 - k_2)^{-1} \mod p) \mod p)$ and $b = (l_1 - ak_1) \mod p$. There is then one-to-one mapping between (a, b) and (l_1, l_2). The number of $h \in H$ such that $h(k_1) = h(k_2)$ is :

$$|\{(l_1, l_2) \mid l_1 \neq l_2 \in \mathbb{Z}_p,\ l_1 = l_2 \mod m\}| \leq \frac{p(p-1)}{|T|} \leq \frac{|H|}{|T|}$$

■

15 Open Addressing

Elements are stored in the table. To insert, we probe the hash table until we find x or an empty slot. If we find an empty slot, insert x here. To define which slots to probe, we use a hash function that depends on the key and the probe number. To search, we probe the hash table until we either find x (return YES) or an empty slot (return NO). In the analysis, we will assume h to be uniform.

Théorème 15.0.1 (Analysis)**.** *Given an open-address hash-table with load factor $\alpha = \frac{n}{|T|} < 1$, the expected number of probes in an unsuccessfulsearch is at most $\frac{1}{1-\alpha}$, assuming uniform hashing.*

Démonstration. An unsuccessful search on x means that every probed slot except the last one is occupied and does not contain x, and the last one is empty. We define A_i the event « The i-th probe occurs and is occupied. ». By Bayes's Theorem, we must estimate :

$$\mathbb{P}[\# \text{ of probes} \geq i] = \prod_{k=1}^{i-1} \mathbb{P}[A_k \mid \bigcap_{j=1}^{k-1} A_j]$$

But we have : $\mathbb{P}[A_j \mid A_1 \cap \ldots \cap A_{j-1}] = \frac{n-j+2}{|T|-j+2}$ So we have : $\mathbb{P}[\# \text{ of probes} \geq i] \leq \frac{n}{|T|}^{i-1} = \alpha^{i-1}$ Then, the expected number of probes is :

$$\sum_{i=1}^{+\infty} \mathbb{P}[\# \text{ of probes} \geq i] \leq \sum_{i=1}^{+\infty} \alpha^{i-1} = \frac{1}{1-\alpha}$$

■

Corollaire 15.0.1.1. *The expected number of probes during insertion is at most $\frac{1}{1-\alpha}$.*

Démonstration. If we insert x we first ran an unsuccessful search for it ■

Théorème 15.0.2. *The expected number of probes during a Successful search is at most $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$.*

Démonstration. A successful search for x probes the same sequence of slots as insertion.
If x is the i-th element inserted into the table, insertion probes less than $\frac{1}{1-\frac{i}{|T|}}$ slots in expectation. Therefore, the expected time of a successful search is at most :

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{|T|}{|T|-i} = \frac{|T|}{n} \sum_{i=0}^{n-1} \frac{1}{|T|-i} = \sum_{i=0}^{|T|} \frac{1}{i} - \sum_{i=0}^{|T|-n} \frac{1}{i} \leq \frac{|T|}{n} \ln \frac{|T|}{|T|-n} = \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

■

This is hard to implement, so we will use heuristics.

15.1 Heuristic hash functions

Let $h^{'}, h^{''}$ be two auxiliary hash functions.

— Linear Probing : $h(k,i) = (h^{'}(k) + i) \mod |T|$ This is easy to implement but it suffers from clustering.

— Quadratic Probing : $h(k,i) = (h^{'}(k) + c_1 i + c_2 i^2) \mod |T|$ We must choose the constants c_1 and c_2 carefully, and this still suffers from clustering.

— Double Hashing : $h(k,i) = (h^{'}(k) + i h^{''}(k)) \mod |T|$ To use the whole table, $h^{''}(k)$ must be relatively prime to m, e.g. $h^{''}(k)$ is always odd, $m = 2^i$.

16 Cuckoo Hashing

Hashing scheme with search time constant in the worst case, as it maintains a hash function without collisions to achieve perfect hashing. This is possible if the set of keys is static. Assume that we have two hash functions h_1, h_2 that satisfy SUHA. We store x in either $T[h_1(x)]$ or $T[h_2(x)]$. Search for x is done in constant time.

```
function INSERT(x)
    if x = T[h1(x)] or x = T[h2(x)] then return
    end if
    pos ← h1(x)
    for i ← 0 to n do
        if thenT[pos] = Null
            T[pos] = x ; return
        end if
        x ⟷ T[pos]
        if pos = h1(x) then
            pos ← h2(x)
        else
            pos ← h1(x)
        end if
    end for
    REHASH
    INSERT(x)
end function
```

Théorème 16.0.1. *This insertion is done in constant time.*

Lemme 16.0.2. *Suppose that $|T| \geq c \cdot n$ for some $c > 1$. For any i, j, the probability that there exists a path from i to j of length $l \geq 1$ which is a shortest path from i to j is at most $\frac{1}{c^l \cdot |T|}$*

Démonstration. By induction on l :

- Initialization : By SUHA, $\mathbb{P}[h_{1,2}(x) = y] = \frac{1}{|T|}$. Thus $\mathbb{P}[\text{there is an edge from } i \text{ to } j] = \frac{n}{|T|^2} \leq \frac{1}{c|T|}$
- Heredity : For $l \geq 1$ there must exist k such that there is a path of length $l-1$ from i to k and an edge from k to j.

■

Démonstration. We define the bucket of x as all the cells that can be reached either from $h_1(x)$ or $h_2(x)$. x, y are in the same bucket if and only if there is a path from $\{h_1(x), h_2(x)\}$ to $\{h_1(y), h_2(y)\}$. Then :

$$\mathbb{P}[x, y \text{ are in the same bucket}] \leq 4 \sum_{l=1}^{\infty} \frac{1}{c^l |T|} = \frac{4}{(c-1)|T|}$$

So :

$$\mathbb{E}[|\text{bucket of } x|] = \mathbb{E} \sum X_{x,y} = \sum \mathbb{E}[X_{x,y}] = \sum \mathbb{P}[x, y \text{ are in the same bucket}] \leq \frac{4}{c-1}$$

Hence, in the absence of rehash, expected insertion time in constant.

The probability that we need a rehash is at most probability that there is a cycle, i.e. a path from i to i : $\frac{4}{c-1} \leq \frac{1}{2}$. The probability that we will need d rehashes is then at most $\frac{1}{2^d}$. Thus, the expected time per insertion is :

$$\frac{1}{n} \cdot \mathcal{O}(n) \sum_{d=1}^{+\infty} \frac{1}{2^d} = \mathcal{O}(1)$$

■

16.1 Rolling Hash Functions

Définition 16.1.1. *The Karp-Rabin fingerprint of a string $S = s_1 \dots s_m$ is :*

$$\varphi(s_1 \dots s_m) = \sum_{i=1}^{m} s_i r^{m-1} \mod p$$

where p is a big enough prime and $r \in \mathbb{F}_p$.

Proposition 16.1.1.
— *If $S = T$, then $\varphi(S) = \varphi(T)$*
— *Else, $\varphi(S) \neq \varphi(T)$ with high probability.*

Démonstration. Let σ be the size of the alphabet, $p \geq max\{\sigma, n^c\}$ where $c > 1$ is a constant. We have :

$$\varphi(S) = \varphi(T) \Leftrightarrow \sum_{i=1}^{m} (s_i - t_i) \cdot r^{m-i} \mod p = 0$$

Hence, r is a root of $P(x) = \sum_{i=1}^{m} (s_i - t_i) \cdot x^{m-i}$ a polynomial over $\mathbb{F}_p$. The probability of such event is at most $\frac{m}{p} \leq \frac{1}{n^{c-1}}$. ■

The Karp-Rabin algorithm is as follows :
— Compute the fingerprint of the pattern
— Compare it with the fingerprint of each m-length substring of the text. If the fingerprint is equal to the fingerprint of a substring, report it as an occurrence

Proposition 16.1.2. *We have :*

$$\varphi(s_1 \dots s_{j+1}) = \varphi(s_1 \dots s_j) \cdot r + s_{j+1} \mod p$$

Démonstration. Observe that :

$$\varphi(s_1 \dots s_{j+1}) = \sum_{i=1}^{j+1} s_i r^{j+1-i} \mod p$$

■

17 TD 3 : Hashing - 13/10

17.1 Exercise 1

We can expect to have $\frac{n(n-1)}{m} - 1$ collisions, i.e. if we consider the sets of pairs of values with collisions, we can expect its cardinality to be $2 \times (\frac{n(n-1)}{m} - 1)$.

17.2 Exercise 2

By definition of h, the sets $\left(h^{-1}(i)\right)_{i \in [1,m]}$ form a partition of U. Thus, since $|U| > nm$, there exists i such that $\left|h^{-1}(i)\right| > n$, giving us the result.

17.3 Exercise 3

We get, by sum, if $k = k_n \dots k_0$, $h(k) = \sum_{i=0}^{n} k_i * (2^p)^i \mod 2^p - 1 = [\sum_{i=0}^{n} k_i] \mod 2^p - 1$. This formula shows the order of the caracters doesn't matter in the value by the hash function.

17.4 Exercise 4

It is sufficient to find the cardinality of the set $\{ih_2(k) \mid i \in [0, m-1]\}$. This is, by Euler's formula m if and only if $h_2(k) \wedge m = 1$

17.5 Exercise 5

17.6 Exercise 6

17.6.1 Question a.

We know that the number of probes during insertion is the number of probes seen during search. An unsuccessful search on x means that every probed slot except the last one is occupied and does not contain x, and the last one is empty. We define A_i the event « The i-th probe occurs and is occupied. ». By Bayes's Theorem, we must estimate :

$$\mathbb{P}[\# \text{ of probes} \geq i] = \prod_{k=1}^{i-1} \mathbb{P}[A_k \mid \bigcap_{j=1}^{k-1} A_j]$$

But we have : $\mathbb{P}[A_j \mid A_1 \cap \ldots \cap A_{j-1}] = \frac{n-j+2}{|T|-j+2}$
So we have : $\mathbb{P}[\# \text{ of probes} \geq i] \leq \frac{n}{|T|}^{i-1} = \alpha^{i-1}$
Here, $\alpha \leq \frac{1}{2}$ thus we have the result.

Then, we get that the probability of the i-th insertion requiring strictly more than $2\log_2(n)$ probes is at most $\frac{1}{n^2}$

17.6.2 Question b.

We have : $\mathbb{P}[X > 2\log_2(n)] = \mathbb{P}\left[\bigcup_{i=1}^{n} X_i > 2\log_2(n)\right] < \sum_{i=1}^{n} \mathbb{P}\left[X_i > 2\log_2(n)\right] = \frac{1}{n}$. The last inequality is strict as these events are not independent.

17.6.3 Question c.

The expected length of the longest probe sequence is :

$$\mathbb{E}[X] \leq \mathbb{P}[X \leq 2\log_2(n)] \times 2\log_2(n) + \mathbb{P}[X > 2\log_2(n)] \times n = \mathcal{O}(\log_2(n))$$

17.7 Exercise 7

17.7.1 Question a.

If H is 2-independent, then for each pair of distinct keys x_1, x_2 and y_1, y_2 is $\mathbb{P}[h(x_1) = y_1, h(x_2) = y_2] = \frac{1}{m^2}$. If we fix $y_1 = y_2$, we have : $\mathbb{P}[h(x_1) = y_1 = h(x_2)] = \frac{1}{m^2}$. By summing over all values for y_1 : $\mathbb{P}[h(x_1) = h(x_2)] = \frac{1}{m}$.
However, as H is universal if and only if : $\mathbb{P}[h(x_1 = h(x_2)) \mid h \leftarrow^s H] \leq \frac{1}{m}$, we have the result.

17.7.2 Question b.

By the same argument as in the class, H is universal because we have a one-to-one mapping between U and H. Moreover, with $x = (0, \ldots, 0) \mod p$, we get the result.

17.7.3 Question c.

Again this family is universal. We have : $\mathbb{P}[h'_{a,b}(x_1) = y_1 \wedge h'_{a,b}(x_2) = y_2] = \frac{1}{m^2}$ since a and b are uniquely determined by $h'(a, b)$.

17.7.4 Question d.

Given the fact that we H is 2-independent, we get the result since this comes from : $\mathbb{P}[h(m) = t \wedge h(m_0) = t_0] = \frac{1}{p}^2$. But, knowing the family, since this family is universal, the adversary might have better knowledge of which functions have same image over m, he can succeed to fool Bob with probability at most $\frac{1}{p}$.

17.8 Exercise 8.

17.8.1 Question a.

This is Langrange's Theorem.

17.8.2 Question b.

As there is an even number of elements in $\mathbb{F}$, there are as many elements in $\mathbb{F}$ ordered with an even and an odd number. Then, by question 1, since there is a one-to-one mapping between $\mathbb{F}^4$ and G, there are, for each tuple in $\{\pm 1\}^4$, $2^4 = m^4$ functions that suit. Thus, H is 4-independent.

Cinquième partie

Lecture 4 : 26/10

18 Binary Search Trees

Définition 18.0.1. *A Binary Tree is a rooted tree with every node having at most two children. Binary search trees are binary trees verifying the following :*

If a node contains element l, the left subtree rooted at the left child contains only elements $\leq l$ and the right subtree elements $> l$.

Proposition 18.0.1. *To access a given element, the time needed is $\mathcal{O}(h)$ time where h is the height of the tree.*

h is not necessarily small, there are many implementations to bound h.

18.1 Red-Black Trees

Définition 18.1.1. *A red-black tree is a binary tree that satisfies the following red-black properties :*

1. *Every node has a color : red or black*
2. *The root is black*
3. *Every leaf (NIL) is black*
4. *If a node is red, both its children are black*
5. *For each node, all paths from the node to descendant leaves contain the same number of black nodes.*

Lemme 18.1.1. *The height of a red-black tree with n nodes is at most $2\log(n+1)$.*

Démonstration. We introduce $bh(x)$ the number of black nodes in a path from x to a leaf.
By induction on $bh(x)$, the subtree rooted at x contains at least $2^{bh(x)} - 1$:

- Initialization : $bh(x) = 0$ implies the tree has more than 0 nodes which is true.
- Heredity :
  - If x is black, let y_1, y_2 denote its children. We have $bh(y_1) = bh(x) - 1 = bh(y_2) - 1$. By summing, we get the result.
  - If x is red, both of its children are black by rule 4. Let us denote the grand children of x by $z_1, \ldots, z_4$. We have : $bh(z_i) = bh(x) - 1$. By sum, we get the result.

Then if h the black height of the tree, $n \geq 2^h - 1$ and $h \leq \log n + 1$. Since in any root-to-leaf path, the number of nodes is at most twice the number of black nodes per rule 4, the lemma follows. ∎

Proposition 18.1.1 (Insertion)**.** *We insert a node into a n-node red-black tree by :*

- *Inserting the node into T as if it were an ordinary binary search tree*
- *We color it red*

— *We perform a number of rotations and node recolouring.*

This is done in $\mathcal{O}(\log n)$.

We define the left rotation on a node x whose right child is not null like this :

$$Tree(x, \alpha, Tree(y, \beta, \gamma)) \mapsto Tree(y, Tree(x, \alpha, \beta), \gamma)$$

We define right rotations to be the reverse operation.

Proposition 18.1.2. *Rotations do not break the BST properties.*

Only on right rotations. Elements in γ are greater than y, in β are greater than x and lower or equal than y, and elements in α are lower or equal than x. After the rotation, we still have those properties. ■

Proposition 18.1.3 (Restoring red-black properties)**.** *Rule* 1 *is never violated. Rules* 2 *and* 3 *can be fixed in* $\mathcal{O}(1)$ *time. Rule* 5 *is never violated as we colour the new node red.*
We will try and maintain the invariant that the properties are broken in at most one node. There are three cases in which Rule 4 *can be violated : Let* z *be the node that violates the red-black properties.*

1. *z's uncle* y *is red* $\Rightarrow$ *We colour z.p black, colour* y *black and z.p.p red. We moved the problematic node up by two layers.*
2. *z's uncle* y *is black and* z *is a right child* $\Rightarrow$ *We left-rotate around the edge z.p-z, we then have z's uncle* y *to be black and* z *to be a left child, i.e. case 3.*
3. *z's uncle* y *is black and* z *is a left child* $\Rightarrow$ *We color z.p black, colour z.p.p red and right-rotate.*

Proposition 18.1.4 (Complexity)**.** *Red black tree use* $\mathcal{O}(n)$ *space,* $\mathcal{O}(\log n)$ *in time for insertion and deletion in the worst case.*

18.2 Treaps

Définition 18.2.1. *Treaps are binary trees with every node having a key and a priority verifying :*

— *a binary search tree for the keys*

— *a min-heap for the priorities*

Proposition 18.2.1 (Search)**.** *The usual algorithm yields time in* $\mathcal{O}(depthofthenode)$ *for a successful search and* $\mathcal{O}(\max(depth(v^-), depth(v^+)))$ *where* $key(v^-)$ *is the predecessor of the searched key.*

Proposition 18.2.2 (Insertion)**.** *The standard BST algorithm with added rotations to fix the heap properties (if* $p(z) \leq p(z.p)$*, rotate auround* $(z, z.p)$*) yields time complexity in* $\mathcal{O}(\max(depth(v^-), depth(v^+)))$.

Proposition 18.2.3 (Deletion)**.** *We run the insertion algorithm backwards : let* $light(z)$ *be the child of* z *with smaller priority. The choice of this edge preserves the heap property everywhere except at* z*. As long as* z *is not a leaft, rotate around* $(z, ligth(z))$*, then chop it off.*

Proposition 18.2.4 (Split)**.** *If we want to split the tree along a pivot* π*, we can insert a new node* z *with key* π *and priority* $-\infty$*. After the insertion, this new node is the root since it has the smallest priority, and all keys in the left subtree are smaller than the pivot and this subtree is a treap.*

Proposition 18.2.5 (Merge)**.** *Suppose that we want to merge two treaps* T_- *and* T_+*, where keys in the first tree are smaller than the keys in the second tree. We create a dummy root with priority* $-\infty$ *and then delete it.*

Proposition 18.2.6 (Complexity)**.** — *Time for a successful search :* $\mathcal{O}(depth(v))$ *where* $key(v) = k$.

— *Time for an unsuccessful search :* $\mathcal{O}(\max(depth(v^-), depth(v^+)))$ *where* $key(v^-)$ *is the predecessor of the searched key.*

— *Insertion Time :* $\mathcal{O}(\max(depth(v^-), depth(v^+)))$ *where* $key(v^-)$ *is the predecessor of the searched key.*
— *Deletion Time :* $\mathcal{O}(tree\ depth)$
— *Split/Merge Time : same as insertion / deletion*

We will choose the priorities to be independently and uniformly distributed continuous random variables. We denote by $x_1, \ldots, x_n$ the nodes of the tree in the increasing order of the keys. We denote by $i \uparrow k$, the event x_i if a proper ancestor of x_k : $\text{depth}(x_k) = \sum_{i=1}^{i=n} [i \uparrow k]$

Lemme 18.2.1. *Define*

$$X(i,k) = \begin{cases} x_i, \ldots, x_k & \text{if } i < k \\ x_k, \ldots, x_i & \text{otherwise.} \end{cases}$$

For all $i \neq k$ we have $i \uparrow k$ iff x_i has the smallest priority in $X(i,k)$.

Démonstration.
1. If x_i is the root, then x_i has the smallest priority
2. If x_k is the root, then x_i is not an ancestor of x_k and does not have the smallest priority
3. If x_j is the root with $i < j < k$, then x_i is not an ancestor of x_k and does not have the smallest priority.
4. If x_j is the root and either $i < k < j$ or $i < j < k$ then x_i and x_j are in the same subtree and the claim follow by induction.

■

Corollaire 18.2.1.1.

$$\mathbb{P}[i \uparrow k] = \begin{cases} \dfrac{1}{k-i+1} & \text{if } i < k \\ 0 & \text{if } i = k \\ \dfrac{1}{i-k+1} & \text{if } i > k \end{cases}$$

Then :

$$\mathbb{E}[depth(x_k)] = \sum_{i=1}^{i=n} \mathbb{P}[i \uparrow k] = \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{k+1}^{n} \frac{1}{i-k+1} = H_k - 1 + H_{n-k+1} - 1 < 2\ln n - 2$$

Thus, all treap operations take $\mathcal{O}(\log n)$ time in expectation

Remarque 18.2.1.1. *The fastest sorting algorithm is Quicksort, and it can be seen as inputing all the values in the list in a treap then taking the infix order of the treap.*

Remarque 18.2.1.2. *Tango trees are $\mathcal{O}(\log\log n)$-competitive and Splay trees are conjecture to be fastest in $\mathcal{O}(1)$-competitive, but we do not know if this is true.*

19 Lower Bound for Sorting

The question here is : can we sort in $\mathcal{O}(n \log n)$ or better ? If comparisons only are allowed then no : indeed, any comparison-based sorting program can be thought of as defining a decision tree of possible executions. Running the same program twice on the same permutation causes it to do the exact same thing bu running it on different permutations of the same data causes a different sequence of comparisons to be made on each.

19.1 Decision Tree

Définition 19.1.1. *A decision tree for a sorting algorithm is a binary tree that shows the possible executions of an algorithm on a set.*

Proposition 19.1.1. *The minimum height of a decision tree is the worst-case complexity of comparison-based sorting.*

Lemme 19.1.1. *The height of any decision tree is* $\Omega(n \log n)$

Démonstration. Since any two different permutations of n elements require a different sequence of steps to sort, there must be at least $n!$ different paths from the root to leaves in the decision tree. Thus there must be at least $n!$ different leaves in this binary tree. Since a binary tree of height h has at most 2^h leaves, we know that : $n! \leq 2^h$ i.e. $h \geq \log(n!)$. Finally, we have $\log n! = \Omega(n \log n)$. ∎

Théorème 19.1.2. *Any comparison-based sorting algorithm must use* $\Omega(n \log n)$ *time.*

20 Predecessor Problem

We want to maintain a set S of integers from a universe $U = [1, u]$ subject to insertions, deletions, predecessor and succesor queries. This problem is harder than dictionaries and hashing. BTSs give a solution in $\mathcal{O}(n)$ space and $\Theta(\log n)$ time.

20.1 van Emde Boas Trees

If the time per operation satisfies : $T(u) = T(\sqrt{u}) + \mathcal{O}(1)$, by Substitution, $T(u) = \mathcal{O}(\log \log u)$. We will split U into $\sqrt{u}$ chunks of size $\sqrt{u}$ size, and the recurse.

Définition 20.1.1 (Recursive van Emde Boas Trees)**.**
- *T.summary is a vEB-tree of size* $\sqrt{u}$ *containing all* i *such that the* i*-th chunk is not empty.*
- *For each* $1 \leq i \leq \sqrt{u}$*, T.chunk[i] is a vEB-tree of size* $\sqrt{u}$ *containing* $x mod \sqrt{u}$ *for each* x *in the* i*-th tree.*
- *T.min is the smallest element in* T*, not stored recursively.*

We represent each integer $x = \langle c, i \rangle$ *where*
- c *is the chunk coordinates :* $c = x // \sqrt{u}$
- i *is the position of* x *in the chunk :* $i = x mod \sqrt{u}$

Proposition 20.1.1 (Successor Operation)**.** *With the following algorithm we get the wanted complexity :*

Algorithme 7 Successor Complexity Verifies : $T(u) = T(\sqrt{u}) + \mathcal{O}(1)$

---

function (Successor)
    Input $(T, x = \langle c, i \rangle)$
    if $x < T.min$ **then**
        return $T.min$
    end if
    if $i < T.chunk[c].max$ **then**
        return $\langle c, \text{Successor}\,(T.chunk[c], i) \rangle$
    else
        $c' = \text{Successor}(T.summary, c)$
        return $\langle c', T.chunk[c'].min \rangle$
    end if
end function

Proposition 20.1.2 (Insertion)**.** *The algorithm below gets the correct complexity.*

Proposition 20.1.3 (Deletion)**.** *The algorithm below gets the correct complextiy.*

Proposition 20.1.4 (Complexity)**.**
- *All of these operations' complexities verify* $T(u) = T(\sqrt{u}) + \mathcal{O}(1)$ *and thus have time complexity* $\mathcal{O}(\log \log u)$
- *Space complexity satifies :* $S(u) = (\sqrt{u} + 1)\,\dot{S}(\sqrt{u}) + \mathcal{O}(1)$ *and therefore* $S(u) = \mathcal{O}(u)$*.*

Algorithme 8 Insertion Complexity Verifies : $T(u) = T(\sqrt{u}) + \mathcal{O}(1)$

function (Insert)
  Input $(T, x = \langle c, i \rangle)$
  if $T.min = None$ **then**
    $T.min = T.max = x$
    return
  end if
  if $x < T.min$ **then**
    swap$(x, T.min)$
  end if
  if $x > T.max$ **then**
    $T.max = x$
  end if
  if $T.chunk[c].min = None$ **then**
    Insert$(T.summary, c)$
  end if
  Insert$(T.chunk[c], i)$
end function

Algorithme 9 Deletion Complexity Verifies : $T(u) = T(\sqrt{u}) + \mathcal{O}(1)$

function (Delete)
  Input $(T, x = \langle c, i \rangle)$
  if $x = T.min$ **then**
    $c = T.summary.min$
    if c = None **then**
      $T.min = None$. **return**
    end if
    $x = T.min = \langle c, i = T.chunk[c].min \rangle$
  end if
  Delete$(T.chunk[c], i)$
  if $T.chunk[c].min = None$ **then**
    Delete$(T.summary, c)$
  end if
  if $T.summary.min = None$ **then**
    $T.max = T.min$
  else
    $c^{'} = T.summary.max$
    $T.max = \langle c^{'}, T.chunk[c^{'}].max \rangle$
  end if
end function

Proposition 20.1.5 (Original van Emde Boad trees)**.** *For Successor(x), as the path from the root to x is monotone, binary searching the path to find the lowest 1 gives us either the predecessor or the successor of x. By storing all nodes in an array of size $\mathcal{O}(u)$ to allow efficient binary search ; a pointer from each node to the maximum and minimum of their subtree ; all the elements as a doubly-linked list, we find the successor and the predecessor of x in $\mathcal{O}(\log\log u)$ time.*
Update is done in $\mathcal{O}(\log u)$ time, since we only need to update the element-to-root path, in $\Theta(u)$ space

20.2 Improvements

20.2.1 x-fast trees

Définition 20.2.1. *In an x-fast tree, we store every root-to-green node (nodes representing an element from the set) path, viewed in binary (left = 0, right = 1), via Cuckoo Hashing.*

Proposition 20.2.1. *Predecessor queries are done in $\mathcal{O}(\log\log u)$ time, updates in $\mathcal{O}(\log u)$ expected amortised time, but this tree only takes $\mathcal{O}(n\log u)$ space.*

Démonstration. To maintain successor and predecessor, we use the vEB-tree algorithm, giving us the same complexity, and same for updates. Yet, since we only store the root-to-element paths which are of $\log u$ length, we need $\mathcal{O}(n\log u)$ space. ■

20.2.2 y-fast trees

Définition 20.2.2. *We maintain elements in groups of size in $\left[\frac{\log u}{4}, 2\log u\right]$. For each group, we build a BST, and we store representatives of the group using an x-fast tree :*

- *If there are fewer than $\frac{\log u}{2}$ elements, we store them in a single BST.*
- *otherwise, suppose we add/delete an element. If a group becomes too large, we split it in two. If a group becomes too small, we merge it with its neighbour, then split if needed.*

Proposition 20.2.2. *Predecessor queries are in $\mathcal{O}(\log\log u)$ time, updates in $\mathcal{O}(\log\log u)$ expected amortised time, since insertion into the x-fast trie happens only once per $\Theta(\log u)$ new elements.*

Démonstration. This comes directly from the definition and the definition of the x-fast trees. ■