Algorithmique

Pierre Aboulker, Paul Jeanmaire, Tatiana Starikovskaya

5 octobre 2023

Table des matières

Première partie

Lecture 1 - 28/09

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 : $\mathcal{O}(1)$

Input L, 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

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 ← ListeVide()
Sort(a, (fun : x, y ↦ x[1] ≤ y[1]))      ▷ On trie les activités par date de fin croissante
s ← PileVide()
Push(a, s)                               ▷ On ajoute une à une les activités de a dans une pile.
while ( dos)
    ac ← Pop(s)
    if ( thenac 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

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 :

Table des matières

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 :

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

However :

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

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

2. Case 2 : We have :

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

However :

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

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

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

■

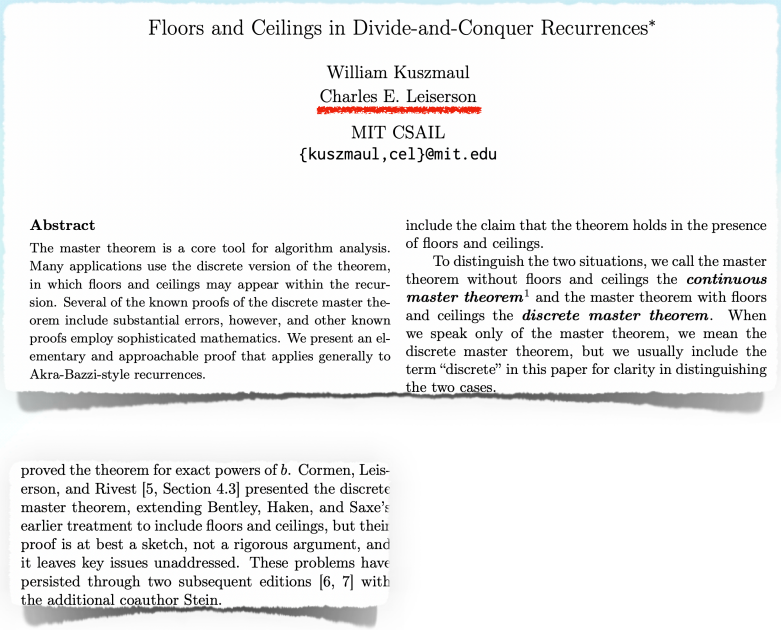
We thus have proved the continuous Master Theorem. ■

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 $L = \Pi_{i=1}^{n}(1 - \frac{1}{\beta^i + 1})^2$, $U = \Pi_{i=1}^{n}(1 + \frac{1}{\beta^i - 1})^2$

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

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

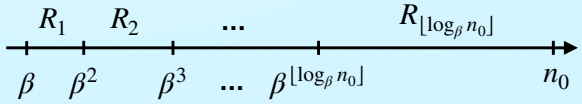
It is enough to show that $c^{-1/4} \leq \Pi_{i=1}^{k}\rho_i \leq c^{1/4}$ for some $c = c(\beta)$

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

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

Proof of Lemma 2 (continued).

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



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

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

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

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

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

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 $c^{-1}(n_0/B) \le n_k \le c(n_0/B)$.

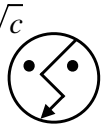
Proof.

Let $c = c(\beta_{\min})$ be the constant from Lemma 3. W.l.o.g. $\sqrt{c} > \max\{\frac{1}{\sqrt{\beta_{\min}} - 1} + 1, \beta_{\min}\}$ $(*)$ 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 $n_{j-1} \ge c^{-1/4}(n_0/B)$

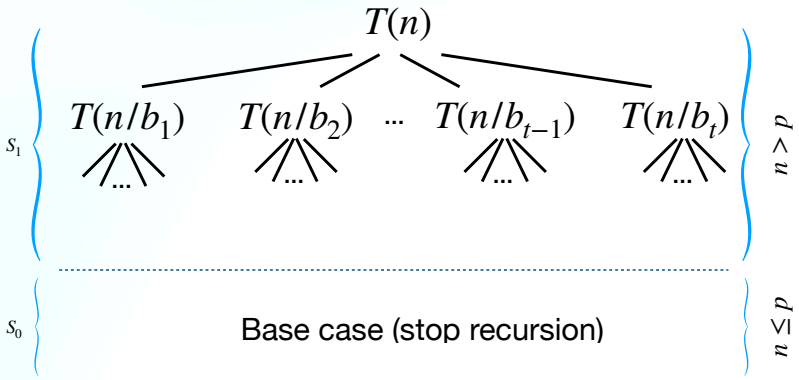
In both cases, $n_{j-1} \ge c^{-1/4}(n_0/B) \ge c^{3/4}$. Therefore, $n_j \ge \lfloor \underbrace{n_{j-1}}_{\ge c^{1/4} > 2\beta_j} / \beta_j \rfloor \ge n_{j-1}/(2\beta_j) > n_{j-1}/c^{1/4} \ge \sqrt{c}$



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 $n \le p$.



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:

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

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

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 $n'_q = \Theta(n_q)$. It follows that $\exists \Phi > 1$ 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:

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

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

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

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

■

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