Issue Date: 29.10.2015
Version: 2015-12-15 17:23

Exercise Problems for Combinatorial Algorithms, WS 15/16

Hand In: *Until the end of the term (or Wed 12:00 for feedback in next meeting), in box (hallway or desk), in person during the meetings, or via email.*

This document provides you with a number of problems designed to help you deepen your understanding of the material. Work on whichever problems you think are interesting and can help you learn. We offer you to review your work and give constructive feedback – all you have to do is hand your solutions in, either in writing or as email.

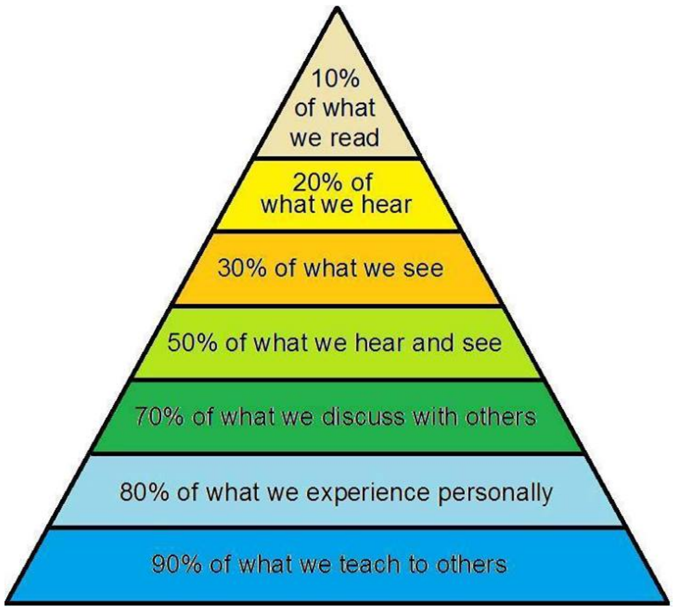
Note that this document is not yet complete and is probably going to be updated frequently. We will notify you whenever we make significant changes. For reference, note the date of this version in the upper right corner.

We list the most recent changes for your convenience:

04.11.2015 Fixes and one addition in chapter Flows & Matchings.

27.11.2015 Creates chapter Symbolic Method and moves problems there.
Changes citation style to be compatible with syllabus.
Adds a final exercise to chapter Compression.

16.12.2015 New warm-up exercise in chapter Symbolic Method.
Problems in chapters Symbolic Method and Random Generation are now up to date.


10% of what we read
20% of what we hear
30% of what we see
50% of what we hear and see
70% of what we discuss with others
80% of what we experience personally
90% of what we teach to others

String Matching

On the Prefix Function

Let $P \in \Sigma^m$. Prove the following properties of prefix function Π_P.

a) **Lemma 6.4 [Neb12]**

For all $q \in \{1, 2, \dots, m\}$,

$$\Pi_P^\star(q) \quad = \quad \{k \mid k < q \wedge P_{0,k} \sqsupset P_{0,q}\} \ ,$$

where $\Pi_P^\star$ is the iterated prefix function (cf. Definition 6.5 in Nebel [Neb12]).

b) **Korollar 6.6 [Neb12]**

For all $q \in \{2, 3, \dots, m\}$,

$$\Pi_P(q) \quad = \quad \begin{cases} 0, & E_{q-1} = \emptyset; \\ 1 + \max E_{q-1}; & E_{q-1} \neq \emptyset, \end{cases}$$

where

$$E_q := \{k \in \Pi_p^\star(q) \mid P_{k+1} = P_{q+1}\}$$

for $q \in \{1, 2, \dots, m-1\}$.

c) Denote with P^r the reverse of P and let $j \in [0..m]$ be arbitrary. If k is the largest natural number *properly* smaller than m with $P_{j+1,m} \sqsupset P_{0,k}$ für $j \in [0..m]$, then

$$\Pi_{P^r}(\ell) \ = \ m - j$$

for $\ell = (m-k) + (m-j)$.

Whenever an exercise states, “Develop an algorithm. . . ” (or similar), your presentation should include a precise description of the algorithm as well as arguments of correctness and an analysis of its (relevant) costs (such as runtime, memory usage, . . .).

Rotation Matching

Develop a linear-time algorithm for the following problem:

Rotation Matching

Input: $A, B \in \Sigma^n$ with $n \in \mathbb{N}$ and some alphabet Σ.

Question: Is there a $k \in \mathbb{N}_0$ so that

$$A_{i+1} \;=\; B_{s(i)+1} \qquad\qquad \text{with } s(i) := (i+k) \bmod n\ .$$

holds for all $i \in [0..n-1]$?

Determine an infinite family of worst-case inputs for your procedure, e. g. by giving a scheme depending on n. How many symbol comparisons does your algorithm need on these inputs?

Regular String Matching

We investigate a generalisation of the *string matching problem.* We now search for a *pattern* (instead of a fixed string P) which we will assume to be given as a *regular language* $L \in \Sigma^*$. We will show that there are efficient algorithms for this problem as well.

You can assume that L be given in one of the usual, finite representations, e. g. finite automata, regular expressions or left-/right-regular grammars[1].

Assume furthermore that L is *fixed*, i. e. the asymptotics in the problem statements below do *not* depend on L resp. the size of its representation. Nevertheless, your algorithms should work for *any* regular L!

a) Develop an algorithm that solves the following problem $\mathcal{O}(n)$ time:

Regular String Matching

Input: $w \in \Sigma^n$

Question: Does w match the pattern L, i. e. is $w \in L$?

b) Develop an algorithm for the following problem:

Regular Substring Matching

Input: Text $T \in \Sigma^N$ and regular language $L \subseteq \Sigma^*$

Output: The set of all substring matches of L in T, i. e.

$$\mathcal{M}_L(T) = \{(i,j) \in [1:n]^2 \mid i \leq j, T_{i,j} \in L\}\ .$$

[1]You remember from your formal language theory course(s) that these can all be derived from each other efficiently.

Your algorithm should run in time $\Theta(n+k)$ where $k = |\mathcal{M}_L(T)|$. Note that since $\Omega(n+k)$ is a trivial lower runtime bound you are looking for an asymptotically runtime-*optimal* algorithm.

Less efficient solutions may yield partial credit depending on how far off they are.

Compression

Compression algorithms have lots of parameters. Unless otherwise noted, you can assume all those parameters to be *arbitrary*[1] *but fixed* that encoder and decoder have to agree upon for correctness.

On Decompositions

Recall the notion of *exhaustive history* as introduced by Lempel and Ziv [LZ76, p 76]. We call the corresponding decomposition of the input word w *LZ77-decomposition (of w).*

Furthermore, Ziv and Lempel define a restricted variant of this decomposition in their proof of Theorem 2 in a later work [ZL78, p 533]. We consider the limit for $n \to \infty$ and call the result *LZ78-decomposition.*

a) Give formal definitions of the LZ77- and LZ78-decomposition for arbitrary $w \in \Sigma^\star$, respectively. Use modern notation like e. g. in the referenced literature [Neb12; SW11].

   What are similarities and differences between the two?

b) Give

   (i) the LZ77-decomposition,

   (ii) the LZ78-decomposition

   (iii) and an arbitrary *non*-exhaustive history

   of $w = aaaaabbababaaabb$.

c) Prove the following claims using the definitions and notation from a):

   (i) Every $w \in \Sigma^+$ has exactly one LZ77-decomposition.

   (ii) Every $w \in \Sigma^+$ has exactly one LZ78-decomposition.

No Free Lunch

Prove the following *no-free-lunch* theorems for lossless compression.

a) For every compression algorithm A and $n \in \mathbb{N}$ there is an input $w \in \Sigma^n$ for which $|A(w)| \geq |w|$, i. e. the "compression" is no shorter than the input.

b) For every compression algorithm A and $n \in \mathbb{N}$,

$$\big|\{w \in \Sigma^{\leq n} : |A(w)| < |w|\}\big| < \frac{1}{2} \cdot \big|\Sigma^{\leq n}\big| ,$$

that is less than half of all inputs of length at most n can be compressed below their original size.

As domain of (all) compression algorithms, we consider the set of (all) injective functions in $\Sigma^\star \to \Sigma^\star$.

The theorems hold for every non-unary alphabet; you can restrict yourself to the binary case, i. e. $\Sigma = \{0, 1\}$, though.

Implementing LZ77

This exercise will lead you towards an efficient (w. r. t. runtime) implementation of the LZ77-decomposition as defined in On Decompositions.

Note that practitioners will want to use constrained versions which get by with a constant amount of memory for the price of worse compression rates.

a) As a first step, consider the following problem:

Longest Prefix Matching

Input: Text $T \in \Sigma^n$, pattern $P \in \Sigma^m$ and index $t \in [1..n]$.

Output: Length $\ell_{\max}$ of the longest prefix of P which occurs in T "before" position t and its matching site, i. e.

$$\ell_{\max} \quad := \quad \max\Big\{\ell \in [0..\min\{n,m\}] \;\Big|\; \exists i \in [1..n-\ell] : i \leq t \wedge T_{i,i+\ell-1} = P_{1,\ell}\Big\}$$

and arbitrary

$$j \in \{i \in [1..n-\ell_{\max}] : i \leq t \wedge T_{i,i+\ell_{\max}-1} = P_{1,\ell_{\max}}\} .$$

Develop an algorithm that solves the Longest Prefix Matching problem in time $\mathcal{O}(t + \ell_{\max})$ using $\mathcal{O}(\ell_{\max})$ memory[2]. Less efficient algorithms may yield partial credit.

Hint: Some algorithms we discussed at the beginning of the course may be a good starting point.

[2]Memory constraints are always meant in addition to the input.

b) Develop an algorithm which computes LZ77(w) for $w \in \Sigma^n$ in time $\mathcal{O}(Cn)$ with $C := |\mathrm{LZ77}(w)|$ the number of phrases in the LZ77-decomposition of w.

   You may use an algorithm as specified in a) as subroutine (even if you did not come up with your own solution).

Bad Cases for LZ Compression

Find bad-case instances for LZ77, LZ78 and LZW, respectively!

Specifically, you are to define

- infinite classes of inputs
- for each parametrization of the respective algorithm
- that result in *bad* compression ratios.

We are looking for a *formal* definition and *symbolic* calculations for the compression ratios!

Lower bounds (as long as they are non-trivial and interesting) and asymptotic results are fine.

Why is this hard? Can you prove that you have found worst-case instances? Why not?

Hint: It may be useful to actually implement the algorithms.

Flows & Matchings

Even and Odd Flows

Let $G = (V, E)$ be a simple graph with integral edge capacities[1] $c : E \to \mathbb{N}$.

Prove or disprove:

a) If all capacities are *even* numbers, i. e. $c(E) \subseteq 2\mathbb{N} = \{2n \mid n \in \mathbb{N}_0\}$, then there is a *maximal* (s,t)-flow f^* with even flow values only, i. e. $f^*(E) \subseteq 2\mathbb{N}$.

b) If all capacities are *odd* numbers, i. e. $c(E) \subseteq 2\mathbb{N} + 1 = \{2n + 1 \mid n \in \mathbb{N}_0\}$, then there is a *maximal* (s,t)-flow f^* with odd flow values only, i. e. $f^*(E) \subseteq 2\mathbb{N} + 1$.

Note: We continue flows $f : E \to \mathbb{R}$ on sets of edges in an *element-wise* fashion, i. e. $f(A) = \bigcup_{e \in A}\{f(e)\}$ for $A \subseteq E$.

Feasible Flows

In applications, we are often not interested in *maximal* flows but rather if a given network admits a certain (additional) amount of flow from certain sources to certain sinks. For example, consider a wastewater system to which we add certain amounts of water (per time) at storm drains and have to move it to treatment facilities.

Formally, we model this as a decision problem:

Feasible Flow

Input: Simple graph $G = (V, E)$ with capacities $c : E \to \mathbb{R}_{\geq 0}$ and *excess* $b : V \to \mathbb{R}$.

Question: Is there a *feasible* flow $f : E \to \mathbb{R}$ with

$$\forall\, v \in V. \qquad b(v) + \sum_{\substack{e \in E \\ e=(u,v)}} f(e) = \sum_{\substack{e \in E \\ e=(v,u)}} f(e) \quad \text{and} \tag{0.1}$$

$$\forall\, e \in E. \qquad 0 \leq f(e) \leq c(e) \quad ? \tag{0.2}$$

[1] Unless otherwise stated, we use *capacity* synonymous to upper capacity bound and assume that no lower capacity bounds are given, i. e. $l(e) = 0$ for all $e \in E$.

We call a node $v \in V$ with positive excess $b(v) > 0$ a *source* and one with negative excess $b(v) < 0$ – i. e. a node with *demand* – a *sink*.

Show that the Feasible Flow problem reduces to the Max-Flow problem. That is, describe an algorithm that solves Feasible Flow by calling an algorithm for Max-Flow as subroutine.

Multi-Machine Scheduling

We consider a certain (class of) *scheduling* problem(s), i. e. the task of assigning "jobs" to "machines" on a discrete time scale so that all jobs finish on time, subject to certain constraints.

Multi-Machine Scheduling with Preemption (MMSP)

Input: Number $m \in \mathbb{N}$ and $T_j = (r_j, p_j, d_j) \in \mathbb{N}^3$ with $d_j \geq r_j + p_j$ for $j \in [1..n]$.

We call r_j the *release time*, p_j the *processing time* and d_j the *deadline* of *job* T_j.

Question: Is there a scheduling of jobs $T_1, \ldots, T_n$ on m identical machines $M_1, \ldots, M_m$, i. e. a mapping $S : \mathbb{N} \times [1..m] \to [0..n]$, which fulfills the following constraints?

i) No job starts early, i. e.

$$\forall\, j \in [1..n], k \in [1..m].\ t < r_j \implies S(t,k) \neq j\ .$$

ii) All jobs finish on time (and are not "overprocessed"), i. e.

$$\forall\, j \in [1..n].\ \sum_{t=r_j}^{d_j} \sum_{k=1}^{m} [S(t,k) = j] = p_j\ .$$

iii) At any given time, at most one machine can process the same job, i. e.

$$\forall\, j \in [1..n], t \in \mathbb{N}.\ \sum_{k=1}^{m} [S(t,k) = j] \leq 1\ .$$

iv) At any given time, every machine can process only one job, i. e. S is indeed a well-defined function.

Output: A schedule that is feasible in the above sense, if there are any.

Note that we have implicitly that

- the machines work synchronously in parallel,

- any machine can process any step of any job and
- jobs may be *preempted* without cost, i. e. processing of any (unfinished) job can be paused at any time and continued on any other machine;

these two properties in particular distinguish MMSP from other, harder scheduling problems.

a) Give a polynomial-time many-one reduction from MMSP to Max-Flow using (at most) logarithmic space (in addition to input and output).

   Note: This is possible because Max-Flow is log-space complete in $\mathcal{P}$ [GSS82].

b) Use the reduction from a) to develop an algorithm for MMSP. What is the (asymptotic) runtime of your algorithm?

c) What is the fastest known algorithm for MMSP? Compare its resource costs (in particular runtime) with the algorithm from b).

   Note: We expect you to do some independent literature research here. Make sure to credit your sources according to academic standards.

Converting Flows to Cuts

The Max-Flow-Min-Cut theorem relates the flow values of maximal flows and the capacities of minimal cuts; the optimal solution *values* of both problems coincide. However, additional work is needed to compute the actual *solutions* — and we only considered flow algorithms in class.

a) Design an algorithm for computing a minimum-capacity s-t-cut in the network $G = (V, E, c)$ from a maximum flow f^* in G.

b) Assume you are given a network $G = (V, E, c)$ and a minimum-capacity s-t-cut. Can you use the cut for determining a maximum flow f^* in G faster than solving Max-Flow from scratch?

Invariants of Preflow-Push

The augmenting-path approach is related to the *primal* Simplex algorithm[2] for solving linear programs (LP): we start with and maintain a feasible solution, which is initially suboptimal. This solution is successively improved until we reach optimality.

In this exercise, we show that the preflow-push approach resembles the *dual* simplex method: we maintain a *dually* feasible solution, which in our case is an s-t-cut. This

[2] For details on LPs and the Simplex algorithm(s) see any textbook on linear optimization, e. g. Hamacher and Klamroth [HK00].

dual solution is modified until it becomes (primally) feasible; here until the preflow becomes a flow.

The original statement of the preflow-push algorithm does not explicitly maintain this cut, therefore we augment the algorithm as follows.

> We maintain a set S of nodes throughout the algorithm, which is initially $S := \{s\}$. The algorithm then invokes a sequence of relabel and push operations.
>
> Each push moves some amount of flow over an edge $(u, v) \in E_f$ of the *residual* network G_f. After each such push-operation, we now check whether $u \notin S$ and $v \in S$, i. e. whether the push was "into S". If yes, we add to S all nodes reachable from u in G_f (including u itself).

Show that the following properties hold:

(i) At any time, $(S, V \setminus S)$ is an s-t-cut.

(ii) The capacity of the cut $(S, V \setminus S)$ never increases.

(iii) If the current pre-flow f fulfills the flow conservation property then its value $\text{val}(f)$ equals the capacity $c(S, V \setminus S)$ of the cut.

Nash Matchings

We consider the following refined bipartite matching problem.

> **Nash Matching**
>
> **Input:** Sets $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$ of players with rankings of the individuals of opposite type.
>
> That is, for each a_i there is a total preference relation[3] $\prec_{a_i} \subseteq B \times B$, and likewise for every b_i, we have the relation $\prec_{b_i} \subseteq A \times A$.
>
> **Output:** A *Nash-matching* of A and B, which we define as a (bipartite) perfect matching $M \subseteq A \times B$ that is *stable*, that is there is *no* pair of players $(a, b) \in A \times B$ for which the following holds:
>
> > There are $a' \in A$ and $b' \in B$ with $a \neq a'$ and $b \neq b'$,
> >
> > (NM 1) $(a, b'), (a', b) \in M$,
> >
> > (NM 2) $b \prec_a b'$ and
> >
> > (NM 3) $a \prec_b a'$.

We say that "c prefers x to y" if (and only if) $x \prec_c y$. Intuitively speaking, a Nash-matching is one given which no two paired individuals agree that it would be preferable to leave each other and form other pairs.

[3]The relations are total orders, i. e. any two elements b, b' are either equal or $b \prec_{a_i} b'$ or $b' \prec_{a_i} b$.

a) Show that there always exists a Nash-matching of A and B by describing an algorithm for constructing such a matching.

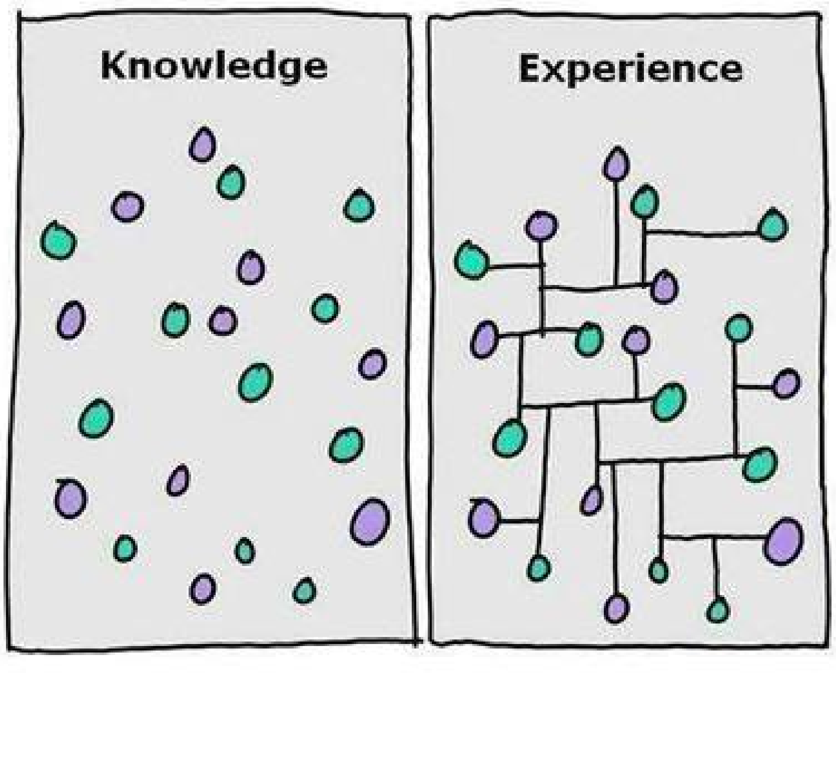
   You may skip the running time analysis of your algorithm, but make sure you prove its correctness.

b) Prove or disprove:
   For all $n \geq 2$ there is a Nash-matching for some preference relation that contains a pair (a, b), where a likes b *least* of all B-players and likewise b prefers all other A-players to a.

c) Prove or disprove:
   For all $n \geq 2$ there is a Nash-matching for some preference relation that contains players $a \in A$ and $b \in B$ which are both paired with their *least* preferred partners, but they are *not* paired with each other.

d) Prove or disprove:
   For all $n \geq 2$ there is a Nash-matching for certain preference relations given which *no* player is paired with its most preferred partner.

e) Prove or disprove:
   For all $n \geq 2$, every preference relation admits exactly one Nash-Matching.



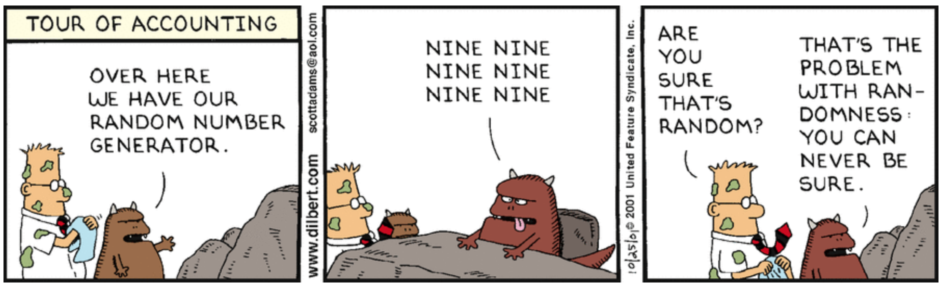
Random Numbers

Several exercises in this chapter are open-ended, and deliberately so. You will have to do some research, form an opinion and present it clearly.

What is Randomness?

Read sections 3.3 and 3.5 of Knuth [Knu01].

a) Make sure you understand the truth, the fallacy and – optionally – the punchline of the following comic strip:


TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
scottadams@aol.com
www.dilbert.com
NINE NINE NINE NINE NINE NINE
© 2001 United Feature Syndicate, Inc.
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.

http://dilbert.com/strip/2001-10-25
© 2011, Universal Uclick

b) Work on a random sample of size $n = 5$ of the given exercises. Hand in the solution you would most appreciate feedback on.

Pseudo- vs True Randomness

Name usage scenarios in which you would prefer true random numbers over pseudo-random numbers, and vice versa. Discuss your reasoning.

Random Number Generation in Practice

a) Discuss the differences of the pseudo-random number generators

   (i) java.util.Random,

   (ii) java.util.SecureRandom and

   (iii) Python's random.py.

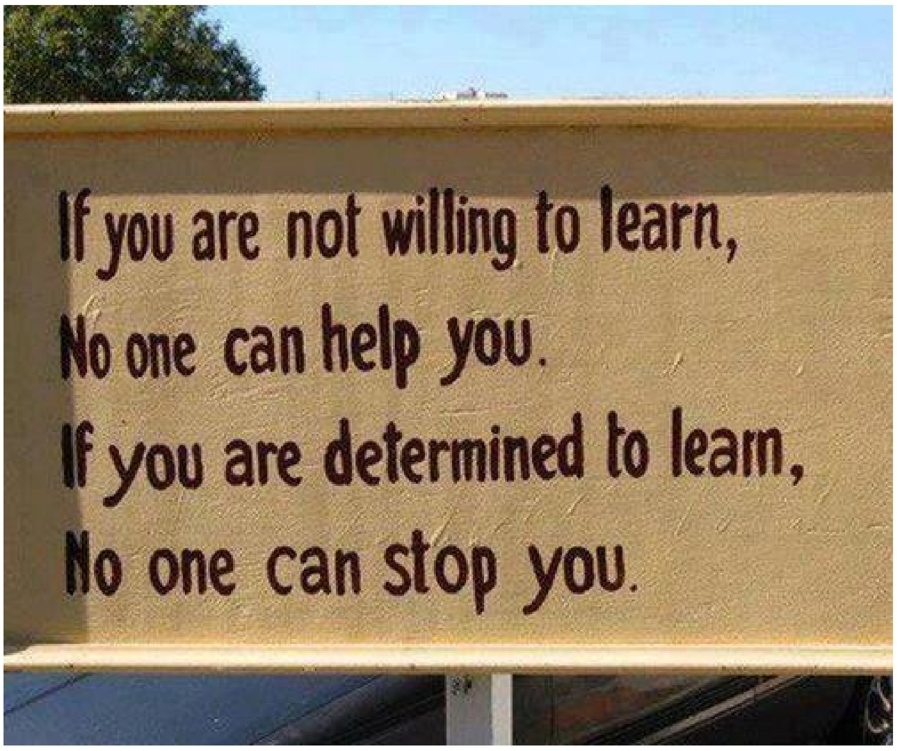
   Note that you are expected to at least skim the sources.

b) Write a *subtly* flawed random number generator; the worse it performs, the better.

   That is, the issue should not be completely obvious; an average programmer skimming the code should be fooled. As a rule of thumb, you may want to make the code look similar to the ones you investigated in a) (or any library PRNG).

   Explain the flaw and the flavor of "bad" your are aiming for.

More exercises on random numbers will be coming. Until then, you can try yourself on the exercise problems in the primary resources.


If you are not willing to learn,
No one can help you.
If you are determined to learn,
No one can stop you.

Symbolic Method

The exercises in this chapter are intended to make you familiar with the use of the *symbolic method* for specifying combinatorial classes. For all exercises of type “Specify …” you are expected to answer the following questions:

- Do we deal with labeled or unlabeled atoms?
- Which atoms do we need and what sizes should they have?
- How can the given structures be constructed (recursively?) from smaller parts?
  - Briefly describe your idea. Pictures are very welcome!
  - Make sure that the specification is complete and unambiguous, i. e. every object has a *unique* construction.
  - Give the (system of) symbolic equation(s).
- What is the generating function for the class, i. e. for the sequence of numbers of objects of each size?

Warmup: Counting

Use the symbolic method to count the following sets of objects.

Hint: You may use the Mathematica function `SeriesCoefficient` (or equivalent functions of other computer algebra systems) for this task. A simple version is available on our website: `http://wwwagak.cs.uni-kl.de/home/lehre/mathe-tools`
(currently only in German; use button „Koeffizient“ in section „Potenzreihenentwicklung“)

a) *Partitions* of $n = 41$, i. e. *multisets* of non-zero natural numbers that sum to n.

For example, $n = 4$ has five partitions, namely

$$4, \qquad 3+1, \qquad 2+2, \qquad 2+1+1, \qquad 1+1+1+1.$$

b) *Compositions* of $n = 41$, i. e. (ordered) *sequences* of non-zero natural numbers that sum to n.

For example, $n = 4$ has eight compositions, namely

$$4, \quad 3+1, \quad 2+2, \quad 2+1+1, \quad 1+3, \quad 1+2+1, \quad 1+1+2, \quad 1+1+1+1.$$

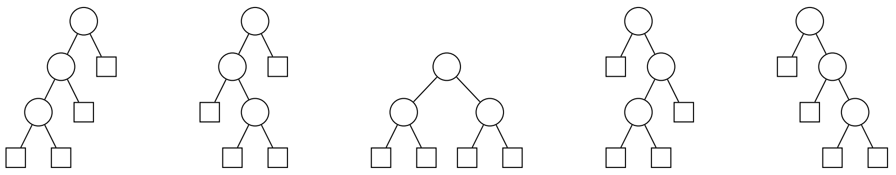
c) Partitions of $n = 41$ with distinct parts, i. e. *sets* of non-zero natural numbers that sum to n.

For example, $n = 4$ has two partitions with distinct parts, namely

$$4, \qquad 3+1.$$

d) *(Extended) binary trees* with 13 inner nodes.

For example, there are five extended binary trees with three inner nodes:



e) *RNA secondary structures* of length 21, modelled as words over the alphabet $\{$`(`, `•`, `)`$\}$, satisfying the following conditions:

(1) The number of opening and closing parentheses is identical.

(2) No prefix of the word contains more closing parentheses than opening ones.

(3) The string `()` does not occur as a substring.

We call a string satisfying (1) and (2) *correctly parenthesized.*

For example, there are the eight structures of length 5:

`•••••`, `(•••)`, `(••)•`, `(•)••`, `•(••)`, `•(•)•`, `••(•)`, `((•))`.

Specifying Trees and Walks

a) Give a formal specification of the class $\mathcal{T}$ of (directed, rooted) trees, in which every node has either 0, 1 or 2 children. In case of a binary node, the order of the subtrees matters. We assume nodes of the same group (same degree) to be indistinguishable.

The size $|t|$ of a tree $t \in \mathcal{T}$ is the total number of nodes in t.

b) Specify the class $\mathcal{P}$ of returning random walks with steps $\nearrow$, $\rightarrow$ and $\searrow$, i. e. paths on the two-dimensional grid $\mathbb{Z}^2$ starting in $(0,0)$ and ending in $(n,0)$ ($n \in \mathbb{N}$). Each single step of such a path can be either the vector $(1,1)$, $(1,0)$ or $(1,-1)$. Moreover, the path may never cross the x-axis, i. e. when the current point is $(k,0)$, we may not make a step $(1,-1)$.

The size of a path is the number of steps, or equivalently its length in x-direction.

Hint: Decompose paths according to the first step.

c) Consider the class $\mathcal{T}$ from a) again, with one difference: this time, we define $|t|$ to be the number of *edges* in the tree. Adapt your specification accordingly.

What can you say about the generating function and thus about the number T_n of trees $t \in \mathcal{T}$ with n edges?

Specifying Classes of Mappings

a) Specify the class $\mathcal{S}_r$ of *surjective* functions in $\{1,\ldots,n\} \to \{1,\ldots,r\}$ for fixed parameter $r \in \mathbb{N}$. The size of a surjection is the size its domain, i. e. $|f| = n$.

b) Specify the class $\mathcal{F}$ of (arbitrary) functions $\{1,\ldots,n\} \to \{1,\ldots,n\}$ with (arbitrary) size $|f| = n$.

Specifying a Formal Language

Specify the class $\mathcal{B}$ of bitstrings $b \in \{0,1\}^\star$ with the following properties:

- b ends with the pattern $P = 01001$.
- P does not occur earlier in b.

Use the number of bits in b as its size.

Specifying with Pointing

Consider the labeled combinatorial class $\mathcal{Q}$ implicitly defined by the specification

$$\Theta\mathcal{Q} \;=\; \mathcal{Q} \star \mathcal{Z} \star \mathcal{Q}\,, \tag{0.1}$$

where Θ is the *pointing operator* as defined by Flajolet et al. [FZV94].

a) Determine the functional equation for the (exponential) generating function $Q(z)$ for $\mathcal{Q}$ according to specification (0.1). Translate this equation of generating functions into an equation of coefficients, thereby deriving a recurrence equation for Q_n, i. e. the number of objects of size n in $\mathcal{Q}$.

b) Note that (0.1) does not specify Q_0. Can you prove a simple closed form of Q_n assuming $Q_0 = 1$?

For an educated guess you might compute the first few entries, say $Q_0, \dots, Q_6$, and ask the *On-Line Encyclopedia of Integer Sequences* for help — but do not forget to prove your claims.

c) Show that $\mathcal{Q}$ is isomorphic to a well-known[1] combinatorial class.

[1] As in, you know it from your own studies.

Random Generation

Sampling Secondary Structures

In this exercise, we consider random generation of RNA secondary structures. RNA is an important single-stranded relative of DNA which exhibits rich folding structures. For details, refer to our lecture *Computational Biology II*.

Here, it suffices to know that secondary structures are isomorphic to *Motzkin words*, which are words over $\Sigma = \{\texttt{(}, \texttt{)}, \texttt{*}\}$ where `(` and `)` form well-nested pairs and `*` can appear anywhere. That is, Motzkin languages are Dyck languages shuffled with $\{\texttt{*}\}^\star$.

For instance, "`*((**(***))*((**))`" is a Motzkin word, but neither "`)(`" nor "`(())(`" are.

Formally, we define the *unlabeled* combinatorial class $\mathcal{S}$ of RNA secondary structures as

$$\mathcal{S} \;=\; \epsilon \;+\; \mathcal{Z}_{\texttt{*}} \times \mathcal{S} \;+\; \mathcal{Z}_{\texttt{(}} \times \mathcal{S} \times \mathcal{Z}_{\texttt{)}} \times \mathcal{S}\;. \tag{0.1}$$

a) Note that Flajolet et al. only consider labeled classes whereas the secondary structures defined by (0.1) are unlabeled. Argue why their approach of random generation is also applicable in our case.

b) Use the method described by Flajolet et al. [FZV94] to design an algorithm that generates a secondary structure of given chosen uniformly among all structures $s \in \mathcal{S}_n$ of size n. The algorithm takes n as input.

   Give at least the following intermediate steps explicitly:

   - the *standard specification* for $\mathcal{S}$,
   - recurrence equations for the number of objects of all classes that occur in the standard specification and
   - the generation procedures for all those classes.

   You may directly simplify the generating procedures where possible because of feature of our special case; make sure to explain your simplifications.

c) Implement your procedure from b) in a programming language of your choice.

   Generate 1000 random structures of size $n = 100$ and draw a histogram of the number of unpaired bases U_n, i. e. the number of `*` atoms. Can you guess the distribution of U_n? Justify your guess and give $\mathbb{E}\, U_n$.

Influence of PRNGs on Combinatoric Sampling

In the series of exercise problems Sampling Secondary Structures, Boltzmann-Sampling of Secondary Structures and Efficient Boltzmann-Sampling of Secondary Structures you have constructed random samplers for the combinatorial class of Motzkin words. We will now investigate your choice of random number generator.

a) Have you used a PRNG?

   If yes, why? Do you expect its distributional characteristics to carry over to the distribution of combinatorial structures you sample; in which way, and why?

b) Does your choice affect the result?

   Experiment! Sample a sizable amount of Motzkin words using different sources of random numbers (at least one pseudo and one "true" source). Come up with several meaningful statistics and check for differences between the sources.

Optional Exercises

The following exercise problems relate to supplementary material only. We include them for your enjoyment.

Asymptotics for Motzkin numbers

In this exercise we consider the *Motzkin numbers* M_n once more. As we have already seen, their (ordinary) generating function is

$$M(z) \;=\; \sum_{n\geq 0} M_n z^n \;=\; \frac{1-z-\sqrt{1-2z-3z^2}}{2z^2}\;. \tag{0.1}$$

We are going to use singularity analysis to derive exact asymptotics for M_n.

a) In order to derive asymptotics, we would like to have a generating function that is as simple as possible. Consider the simpler relative of $M(z)$

$$\hat{M}(z) \;=\; \sum_{n\geq 0} \hat{M}_n z^n \;=\; -\tfrac{1}{2}\sqrt{1-2z-3z^2}\;.$$

Prove that for $n \geq 2$, we have $\hat{M}_n = M_{n-2}$.

b) Derive exact asymptotics for $\hat{M}_n$, i. e., find an explicit expression $\hat{m}(n)$ with

$$\lim_{n\to\infty} \frac{\hat{m}(n)}{\hat{M}_n} \;=\; 1\;.$$

We abbreviate that as $\hat{M}_n \sim \hat{m}(n)$ as $n \to \infty$ and say "$\hat{M}_n$ is *asymptotically equivalent* to $\hat{m}(n)$."

Compute and/or plot the relative error of your asymptotic for some moderate values of n; use your favorite computer algebra system, the online tools on our website `wwwagak.cs.uni-kl.de/mathe-tools.html` or Wolfram Alpha.

Hint: Use the Corollary[1] from Theorem 5.5 of [SF13].

[1]Beware of the typing error in the book: You have to replace ρ^n by ρ^{-n}.

Hint: To compute specific values of the Gamma function $\Gamma(z)$, the following basic properties are handy, see [Old+]:

$$\begin{aligned} \Gamma(n+1) &= n! && n \in \mathbb{N} && (\Gamma 1) \\ \Gamma(z+1) &= z\,\Gamma(z) && z \in \mathbb{C} && (\Gamma 2) \\ \Gamma(\tfrac{1}{2}) &= \sqrt{\pi} && && (\Gamma\tfrac{1}{2}) \\ \Gamma(z)\,\Gamma(1-z) &= \frac{\pi}{\sin(\pi z)} && z \in \mathbb{C} && (\Gamma 3) \end{aligned}$$

c) Recall Sampling Secondary Structures, where you built a top-down random sampler for the combinatorial class $\mathcal{S}$ of RNA secondary structures, given by:

$$\mathcal{S} \;=\; \epsilon \;+\; \mathcal{Z}_{*} \times \mathcal{S} \;+\; \mathcal{Z}_{(} \times \mathcal{S} \times \mathcal{Z}_{)} \times \mathcal{S}\;. \tag{0.2}$$

Use your new skills in singularity analysis to verify or disprove your conjecture about the distribution of the number of unpaired bases in a uniformly chosen RNA secondary structure of size n.

Hint: The top-down sampler identifies a single point where it is decided which sort of atom to produce next. Find this point in your sampler and express the probability p_{*} for a next symbol to be of type $\mathcal{Z}_{*}$ as the ratio of the coefficients of two generating functions. With a computation very similar to b) you can compute the limit of p_{*}.

Boltzmann-Sampling of Secondary Structures

Consider once again the class of secondary structures given in (0.4).

a) Implement a Boltzmann sampler $\Gamma S(x)$ for $\mathcal{S}$ with parameter $x = 0.33$ as described in Section 3 of [DFLS04].

Keep your implementation adaptable for other choices of x, but for simplicity, you may precompute the needed constants externally and hard-code them into your program.

b) Let N be the (random!) size of a RNA secondary structure generated by $\Gamma S(0.33)$. Compute the expected size $\mathbb{E}\,N$ and its standard deviation $\sigma = \sqrt{\mathbb{V}\,N}$.

Use *Chebychev's inequality* to compute an upper bound $N_{0.99}$, such that with at least 99 % probability, a random structure generated by $\Gamma S(0.33)$ has size at most N; formally

$$\Pr[N \le N_{0.99}] \ge 0.99\;.$$

c) Use your Boltzmann sampler to generate 1 000 random RNA secondary structures and draw a histogram of their sizes.

What can you say regarding the Chebychev tail bound you derived in b)?

Figure 1 of [DFLS04] shows three categories of size distributions for Boltzmann samplers: “bumpy”, “flat” and “peaked”. In which of these three categories does $\Gamma S(x)$ seem to belong?

Asymptotics

Let $f(z)$ be an analytic function with convergence radius strictly larger 1. Show that

$$[z^n]f(z)\,\ln\Big(\frac{1}{1-z}\Big) \;\sim\; \frac{f(1)}{n}\,. \tag{0.3}$$

Improved Boltzmann Yields Good Size With High Probability

Prove Theorem 6.1 by Duchon et al. [DFLS04], that is show that

$$\Pr\big[N \in n(1 \pm \varepsilon)\big] \;\to\; 1 \qquad \text{as } n \to \infty\,,$$

where N is the random size of an object returned by $\mu C(x_n; n, \varepsilon)$.

Here, we write $n(1 \pm \varepsilon)$ for short and mean the interval $\big[n(1-\varepsilon), n(1+\varepsilon)\big] \subset \mathbb{R}$.

Efficient Boltzmann-Sampling of Secondary Structures

Consider once again the class $\mathcal{S}$ of RNA secondary structures, given by

$$\mathcal{S} \;=\; \epsilon \,+\, \mathcal{Z}_{*} \times \mathcal{S} \,+\, \mathcal{Z}_{(} \times \mathcal{S} \times \mathcal{Z}_{)} \times \mathcal{S}\,, \tag{0.4}$$

and the Boltzmann sampler $\Gamma S(x)$ you built in Boltzmann-Sampling of Secondary Structures. In this exercise, we will tweak your sampler for efficiency.

a) Check whether the Boltzmann model of RNA secondary structures fulfills the *Mean Value Condition* and the *Variance Condition* [DFLS04, equations (6.1) and (6.3)].

   What guarantees does Theorem 6.1 of the same article provide for your sampler?

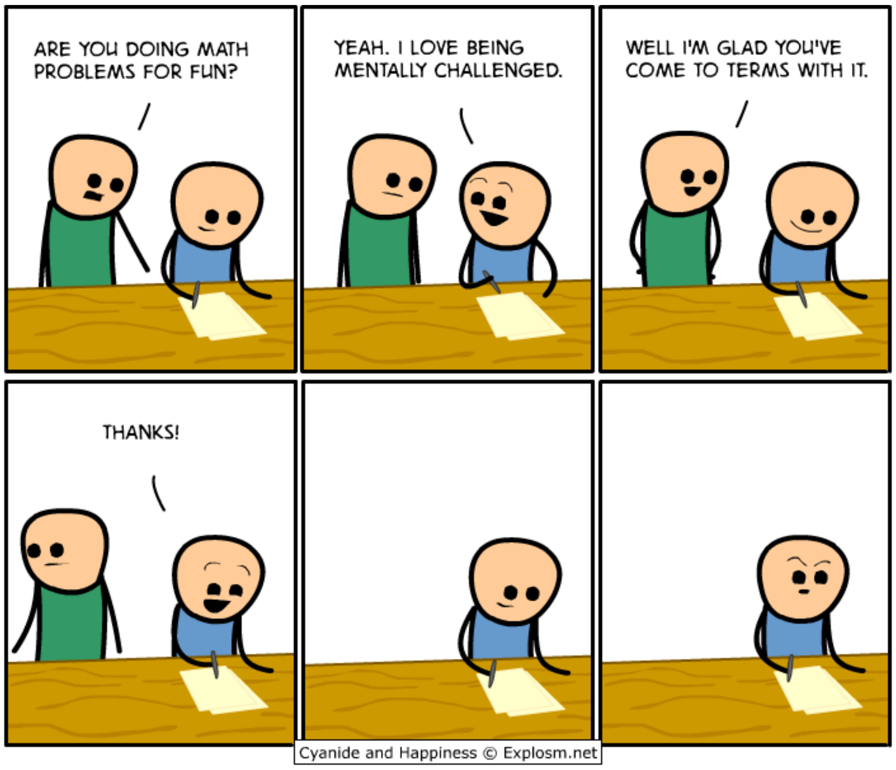
b) Determine the singular exponent $-\alpha$ for $S(z)$ as defined in Section 6.2 [DFLS04].

   What guarantees does Theorem 6.3 of the same article provide for your sampler?

c) Design a linear time approximate size Boltzmann sampler $\mu S(x; n, \varepsilon)$; remember to prove your claims.

   Extend your implementation from Boltzmann-Sampling of Secondary Structures to incorporate this sampling algorithm. For simplicity, you may fix $n = 100$ and precompute all necessary constants externally.

   Use your sampler to draw 10 random RNA structures of size *exactly* 100. How many rejections did your sampler need until it found an object of correct size?


ARE YOU DOING MATH PROBLEMS FOR FUN?
YEAH. I LOVE BEING MENTALLY CHALLENGED.
WELL I'M GLAD YOU'VE COME TO TERMS WITH IT.
THANKS!
Cyanide and Happiness © Explosm.net

Bibliography

[DFLS04] Philippe Duchon et al. “Boltzmann Samplers for the Random Generation of Combinatorial Structures.” English. In: *Combinatorics, Probability and Computing* 13.4-5 (July 2004), pp. 577–625. ISSN: 1469-2163. DOI: 10.1017/S0963548304006315.

[FZV94] Philippe Flajolet, Paul Zimmerman, and Bernard Van Cutsem. “A calculus for the random generation of labelled combinatorial structures.” In: *Theoretical Computer Science* 132.1-2 (Sept. 1994), pp. 1–35. ISSN: 03043975. DOI: 10.1016/0304-3975(94)90226-7.

[GSS82] Leslie M. Goldschlager, Ralph A. Shaw, and John Staples. “The maximum flow problem is log space complete for P.” In: *Theoretical Computer Science* 21.1 (1982), pp. 105–111. ISSN: 0304-3975. DOI: 10.1016/0304-3975(82)90092-5.

[HK00] Horst W. Hamacher and Kathrin Klamroth. *Lineare und Netzwerk-Optimierung / Linear and Network-Optimization. Ein bilinguales Lehrbuch. A bilingual textbook.* Wiesbaden: Vieweg+Teubner Verlag, 2000. ISBN: 9783528031558. DOI: 10.1007/978-3-322-91579-5.

[Knu01] Donald E. Knuth. *Seminumerical Algorithms.* 3rd ed. Vol. 2. The Art Of Computer Programming. Addison-Wesley Longman Publishing, 2001. 762 pp. ISBN: 0201896842.

[LZ76] Abraham Lempel and Jacob Ziv. “On the Complexity of Finite Sequences.” In: *Information Theory, IEEE Transactions on* 22.1 (1976), pp. 75–81. DOI: 10.1109/TIT.1976.1055501.

[Neb12] Markus E. Nebel. *Entwurf und Analyse von Algorithmen.* Wiesbaden: Vieweg+Teubner Verlag, 2012. ISBN: 9783834819499. DOI: 10.1007/978-3-8348-2339-7.

[Old+] Adri B. Olde Daalhuis et al., eds. *NIST Digital Library of Mathematical Functions.* Version 1.0.6. Online companion to the Handbook of Mathematical Functions. URL: http://dlmf.nist.gov/ (visited on 05/06/2013).

[SF13] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms.* 2nd ed. Addison-Wesley Professional, 2013, p. 592. ISBN: 032190575X.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms.* Addison-Wesley, Mar. 2011. ISBN: 9780321573513.

[ZL78] Jacob Ziv and Abraham Lempel. "Compression of individual sequences via variable-rate coding." In: *Information Theory, IEEE Transactions on* 24.5 (1978), pp. 530–536. DOI: `10.1109/TIT.1978.1055934`.