Algorithms and Data Structures
Spring 2018
Exercises for Units 17 and 18, Solutions

Problem 1

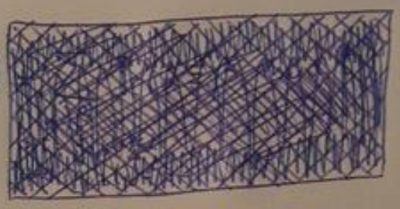
(1.) $h_{a,b}(x) = (a \cdot x + b \bmod p) \bmod t$

$\mathcal{H} = \{h_{a,b} \mid 0 < a < p,\ 0 \le b < p\}$

$Pr(h_{a,b}(x) = h_{a,b}(y)) = Pr((ax+b) \bmod p \equiv (ay+b) \bmod p \bmod t) = (*)$

$g_{a,b}(x) := (ax+b) \bmod p$

$(*) = Pr(g_{a,b}(x) \equiv g_{a,b}(y) \bmod t)$



$$= \sum_{\substack{\alpha,\beta \\ \text{s.t.} \\ \alpha \equiv \beta \bmod t}} Pr(g_{a,b}(x) = \alpha \wedge g_{a,b}(y) = \beta) = (\star)$$

Claim: $Pr(g_{a,b}(x) = \alpha \wedge g_{a,b}(y) = \beta) = \begin{cases} 0, & \alpha = \beta \\ \frac{1}{p(p-1)}, & \text{otherwise} \end{cases}$

(1) $ax + b = \alpha$ (2) $ay + b = \beta$

$ax + \beta - ay = \alpha \longleftarrow b = \beta - a \cdot y$

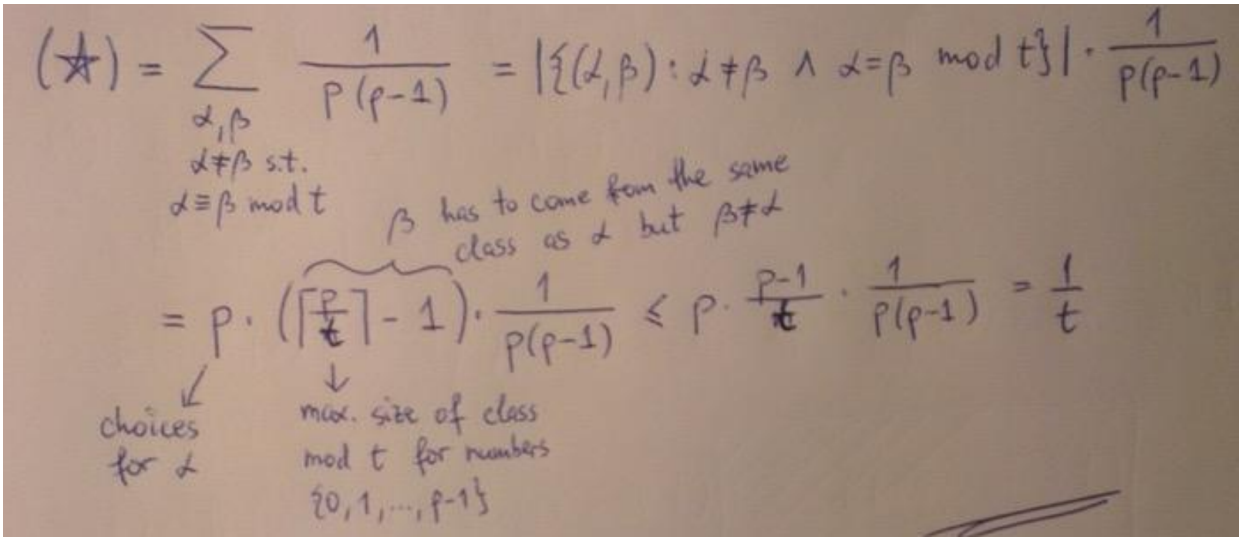
$\boxed{a = (\alpha - \beta)(x-y)^{-1}}$ $\boxed{b = \beta - (\alpha - \beta)(x-y)^{-1} \cdot y}$

$\Rightarrow$ when we take some α, β and want $g_{a,b}(x) = \alpha$ and $g_{a,b}(y) = \beta$, this happens for only one choice of a, b — namely a, b that we can calculate from the formulas above

- for $\alpha = \beta$, $a = 0$ and by assumption $a > 0 \rightsquigarrow$ we cannot have $g_{a,b}(x) = \alpha$ and $g_{a,b}(y) = \beta = \alpha$
- when $\alpha \neq \beta$, only one $g_{a,b}$ is "good" - meaning (1) and (2) are satisfied, and there are $p \cdot (p-1)$ functions in total (p choices for b and $p-1$ for a)

∎

Algorithms and Data Structures
Spring 2018
Exercises for Units 17 and 18, Solutions



Problem 2

1. Let $\mathcal{H}$ be a pairwise independent set of functions, and let $h \in \mathcal{H}$ be selected uniformly at random. Then for all $x \neq y \in U$,

$$\begin{aligned} P[h(x) = h(y)] &= P\left[\bigcup_{i=1}^{t} \{h(x) = i \wedge h(y) = i\}\right] \\ &\leq \sum_{i=1}^{t} P[h(x) = i \wedge h(y) = i] \\ &\leq \sum_{i=1}^{t} \frac{1}{t^2} \\ &= \frac{1}{t} \end{aligned}$$

   since h was selected uniformly at random, for any pair of keys $x \neq y$, there are at most $|\mathcal{H}|/t$ functions $h \in \mathcal{H}$ such that $h(x) = h(y)$; so $\mathcal{H}$ is universal.

2. The converse is not true. Consider the following counterexample: the universe U consists of the first n natural numbers, the table size is n, and the set $\mathcal{H}$ consists of all the perfect hash functions for U, i.e. there are $n!$ functions, each of which is essentially a permutation of U. Clearly $\mathcal{H}$ is universal, since it produces no collisions. But for $i \neq j$, selecting $h \in \mathcal{H}$ uniformly at random,

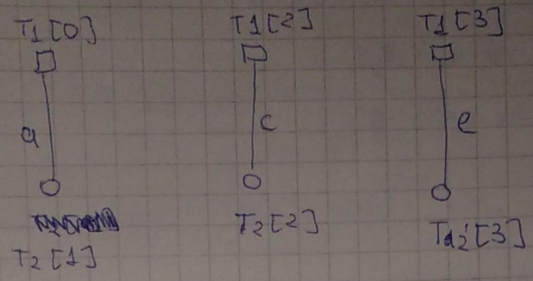
$$\begin{aligned} P[h(x) = i \wedge h(y) = j] &= \frac{(n-2)!}{n!} \\ &= \frac{1}{n(n-1)} \\ &> \frac{1}{n^2} \end{aligned}$$

   so $\mathcal{H}$ is not pairwise independent.

Problem 3

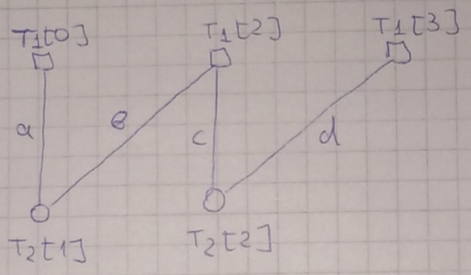
In the following solution $T_1[0..3]$ corresponds to $T[0..3]$ and $T_2[0..3]$ corresponds to $T[4..7]$.

2. $S_1 = \{a, c, e\}$



edges of a, c and e are independent.
We have 2 ways of storing each of a, c and e => number of ways in which we can store S_1 is $2^3 = 8$

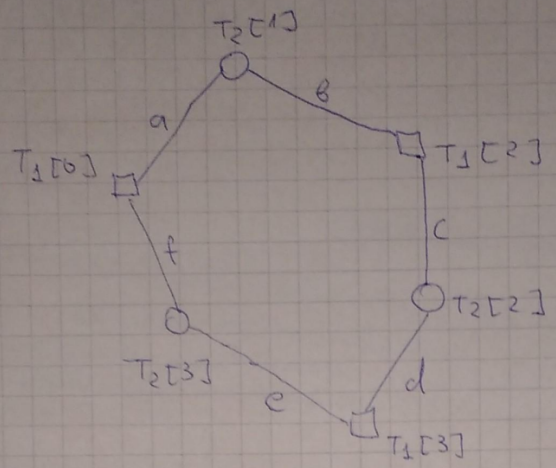
$S_2 = \{a, b, c, d\}$



We have 4 edges and 5 vertices in the graph => one of the vertices must be empty (nothing is stored there).
We have 5 ways of choosing empty vertex and for each choice we have the unique way of storing a, b, c, and d (For example, if $T_1[2]$ is empty, b must be stored in $T_2[1]$, a must be stored in $T_1[0]$, c must be stored in $T_2[2]$ and d must be stored in $T_1[3]$)
So, there are 5 ways of storing S_2.

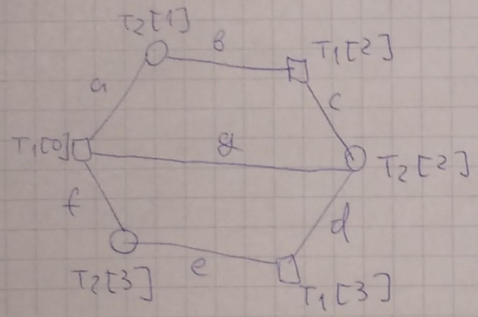
Algorithms and Data Structures
Spring 2018
Exercises for Units 17 and 18, Solutions

$S_3 = \{a, b, c, d, e, f\}$



Graph induced by S_3 is a cycle.
We have 2 ways of storing a and for each of them there is the unique way of storing b, c, d, e and f => we have 2 ways of storing S_3.

$S_4 = \{a, b, c, d, e, f, g\}$



We have more edges than vertices in the Graph =>
=> We can not store S_4.

Problem 4

A first idea is to simply rebuild the entire heap after n creDeaseKey operations thereby removing hollow nodes. This would take $O(n)$ time, which amortized over the in DecreaseKey operations means constant amortized time. However, this straightforward method causes the constant time for DecreaseKey to be valid in the amortized sense only and not in the worst case sense, which was true initially.
How about waiting until the next Delete or MinDelete operation, which would then be allowed to take that much amortized time. The problem is that during the wait a superlinear number of DecreaseKey operations may happen, upping the memory usage in a superlinear way. The obvious way around this is to observe, that when a DecreaseKey operation is applied to a node in the root list, then there is really no need to make the node hollow and move the item to a new node. You can simply decrease the key (and of course compare with and possibly update the minpointer). This way between two delete operations at most n extra nodes can be created, and the restructuring can wait until the next delete operation.

Problem 5

The answer is NO!
Because otherwise you could sort n numbers using comparisons in linear time as follows: build a hollow heap for those number. Then repeat the following n times: do a FindMin (which only need constant time), report that number, and then do an IncreaseKey of that item to $+\infty$ (which supposedly only takes constant time also).

Problem 6

This question was a mistake.
One interesting observation is, though, that linking can be done in a reasonable way in $O(k)$ time. You want of course, that all keys stored in a node are larger than all keys in all the nodes of the subtree. This can be maintained during linking of two nodes by considering the union of the keysets of the two nodes, and storing the smallest k of these $2k$ keys with the winner node, and the other k with the loser node. Using fast median finding, these two sets can be computed in $O(k)$ time.