Institut for Matematik og Datalogi
Syddansk Universitet, Odense

12. december 2018
KSL/RF

Eksaminatorier DM534 Uge 50

Cryptology

1. Is one of the following the multiplicative inverse of 49 modulo 221? Or does no multiplicative inverse exist?

   12, 56, 121, 212

2. Which of the following is a valid RSA key (ignoring the fact that the numbers are not large enough for security)?

   (a) $PK = (91, 37)$; $SK = (91, 23)$
   (b) $PK = (143, 77)$; $SK = (143, 53)$
   (c) $PK = (231, 59)$; $SK = (231, 47)$
   (d) $PK = (107, 25)$; $SK = (107, 30)$

3. Suppose a public RSA key is $PK = (1517, 13)$. Which of the following is the RSA encryption of the message 43?

   (a) $1517^{43} \pmod{13}$
   (b) $43^{13} \pmod{1517}$
   (c) $13^{43} \pmod{1517}$

   For the right answer, we want to use the algorithm for fast modular exponentiation (page 33 on the slides). How many times during the recursive execution is the "**if** k is odd" case encountered, and how many times is the "**if** k is even" case encountered? [Do not include the base cases $k = 0$ and $k = 1$ in the counts.]

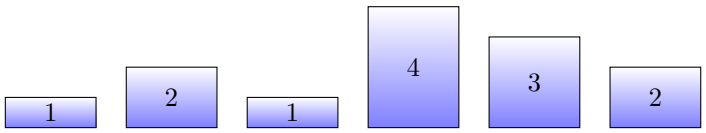
4. In the Sieve of Eratosthenes, how many lists (including the current) have been created at the point where the number 13 is the first element in a list?

Online Algorithms

5. Prove that no matter which other algorithm than the one from the lecture notes we define for ski rental, the algorithm will perform worse, i.e., the competitive ratio will be strictly higher than $\frac{19}{10}$.

   Start by analyzing the algorithms "Buy on day 5" and "Buy on day 15" to see what happens. The skis still cost 10 units to buy and 1 unit per day to rent.

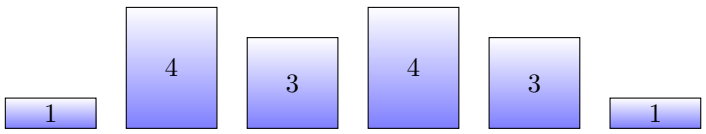
6. For $m = 3$, which schedule does the List Scheduling algorithm, Ls, produce on the following input sequence:



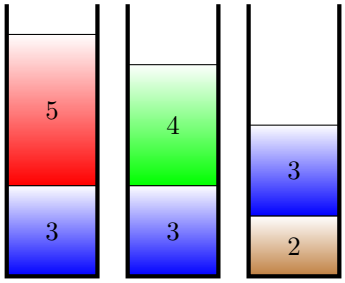
7. In the lecture, we proved that the machine scheduling algorithm, Ls, could not perform better than $2 - \frac{1}{m}$. We now consider only two machines. Thus, $m = 2$, and the ratio is then $\frac{3}{2}$. Just because Ls cannot perform better, it could be that some other algorithm could. Prove (for $m = 2$) that this is not the case. You must design an input, where *no* algorithm, no matter what decisions it makes, can do better than $\frac{3}{2}$ times Opt. You only need sequences with two and three jobs and a case analysis with only two cases, depending on what an algorithm does with the second job that is given.

8. Consider the first bin packing example given in the lecture (slide 19), where the First-Fit algorithm, Ff, uses four bins. Show that Opt only needs three.

9. How does the First-Fit algorithm, Ff, behave on the input sequence below? Item sizes are given in multiples of $\frac{1}{6}$.



10. Why can the following configuration *not* have been produced by the bin packing algorithm Ff? Item size are given in multiples of $\frac{1}{9}$.



11. For bin packing, one can prove the upper bound that FF is 1.7-competitive. However, this is a quite hard proof. In this exercise, we will try to improve (raise) the lower bound.

    In the lecture, we saw an example demonstrating that FF can be as bad as $\frac{3}{2} = 1.5$ times OPT.

    Let that example inspire you, and try to use items of the following three sizes:

    $$\frac{1}{7} + \frac{1}{1000}, \quad \frac{1}{3} + \frac{1}{1000}, \quad \frac{1}{2} + \frac{1}{1000}$$

    Find a sequence where FF performs $\frac{5}{3}$ times worse than OPT.

    Now try using

    $$\frac{1}{43} + \frac{1}{10000}, \quad \frac{1}{7} + \frac{1}{10000}, \quad \frac{1}{3} + \frac{1}{10000}, \quad \frac{1}{2} + \frac{1}{10000}$$

    to get a lower bound close to the 1.7 upper bound.

12. It is very easy to implement FF in JAVA, if there are no efficiency requirements: just use an array to hold the current level in the bins, and for each item, search for the first bin with enough space. If you make sure there are enough bins from the beginning, then there are no special cases. And you simple count the number of non-empty bins at the end to get the result.

    Implement FF.

    Try to define your own algorithm, from scratch or as a variant of FF.

    Test your own algorithm up against FF and try to determine which one is best; for instance on uniformly distributed sequences.

    In Java, one way to create pseudorandom floating point numbers uniformly distributed in the interval $[0, 1[$ is via the class `java.util.Random` and its `NextFloat` method. You may want to look at the tutorial

here: `http://www.functionx.com/java/Lesson18.htm`. If you want to generate the same sequence of pseudorandom numbers in different invocations of your program (for instance to compare two algorithms on the same input sequences), you must set the seed in the random number generator at the start of the program (otherwise it is set to a different seed at each invocation). This is described in the second section of the tutorial.