Prof. Friedrich Eisenbrand
Gennady Shmonin

Due Date: October 6, 2008

Combinatorial Optimization

Fall 2008

Assignment Sheet 1

Exercise 1 (Dual linear programs)
In the lecture we proved that if both a linear program

$$\max\{c^{\mathrm{T}}x : Ax \le b\}$$

and its dual linear program

$$\min\{y^{\mathrm{T}}b : y^{\mathrm{T}}A = c^{\mathrm{T}}, y \ge 0\}$$

are feasible and bounded, then they have the same optimum values.

(a) Mark (and argue!) the entries of the following table corresponding to possible outcomes in this primal/dual pair of linear programs ("bounded" means "feasible and finite"):

	infeasible	unbounded	bounded
infeasible			
unbounded			
bounded			

(b) The above-stated form is not the only possible formulation of a linear programming problem. Write the duals for the following linear programs and show that the strong duality theorem is also valid for them.

- $\max\{c^{\mathrm{T}}x : Ax \ge b\}$
- $\max\{c^{\mathrm{T}}x : Ax = b, x \ge 0\}$
- $\max\{c^{\mathrm{T}}x : Ax \ge b, x \ge 0\}$
- $\max\{c^{\mathrm{T}}x : Ax \le b, x \ge 0\}$
- $\max\{c^{\mathrm{T}}x : Ax = b, x \le 0\}$
- $\max\{c^{\mathrm{T}}x : Ax \ge b, x \le 0\}$
- $\max\{c^{\mathrm{T}}x : Ax \le b, x \le 0\}$
- $\min\{c^{\mathrm{T}}x : Ax \ge b, x \ge 0\}$

Exercise 2 (Feasibility and optimality)
In the previous exercise, we have not listed among the possible forms of linear programming problems the following problem:

$$\max\{c^{\mathrm{T}}x : Ax = b\}.$$

Why is this problem different? Describe a polynomial algorithm to solve the problem.

Exercise 3 (Feasibility and optimality)
Suppose that we have an efficient algorithm to decide if a system of linear inequalities $Ax \le b$ has a feasible solution. Design an efficient algorithm to solve an optimization problem

$$\max\{c^{\mathrm{T}}x : Ax \le b\}.$$

Exercise 4 (Fourier–Motzkin elimination)
Apply Fourier–Motzkin elimination procedure to decide if the following system of linear inequalities has a feasible solution:

$$\begin{array}{rcrcrcrcr} 2x_1 & + & 6x_2 & + & 2x_3 & - & x_4 & \leq & 1 \\ x_1 & - & 2x_2 & + & 3x_3 & + & x_4 & \leq & 0 \\ -3x_1 & + & 6x_2 & - & 2x_3 & - & 3x_4 & \leq & 2 \\ -x_1 & - & 3x_2 & - & 3x_3 & - & 3x_4 & \leq & -3 \\ & & 3x_2 & + & x_3 & + & x_4 & \leq & 2 \end{array}$$

Exercise 5 (Complexity of Fourier–Motzkin elimination)
Show that Fourier–Motzkin elimination procedure for deciding if a given system of linear inequalities has a feasible solution may take exponential (in the number of variables) time. Moreover, it may construct an exponential number of *non-redundant* inequalities.

Hint. Consider the so-called *dual cube* in $\mathbb{R}^n$, which is defined as the convex hull of the unit vectors $e_1, e_2, \ldots, e_n$ and their opposites $-e_1, -e_2, \ldots, -e_n$. On the other hand, the dual cube can be expressed by a system of (how many?) inequalities $Ax \leq b$.

Exercise 6 (Analysis of algorithms)
In this exercise we analyze the performance of some well-known *sorting algorithms*. Each of these algorithms accepts as input an array $A[1 \ldots n]$ of non-negative integers and sorts A in a non-decreasing order.

(a) The algorithm BUBBLESORT looks at the pairs of consecutive elements $A[i]$ and $A[i+1]$, and swaps $A[i]$ and $A[i+1]$ if they are in the wrong order.

BUBBLESORT(A)
Input: an array of integers $A[1 \ldots n]$
do
  swapped := *false*
  for $i := 1$ **to** n **do**
    if $A[i] > A[i+1]$ **then** { SWAP($A[i], A[i+1]$); *swapped* := *true*; }
while *swapped* = *true*

Implement (in pseudo-code) the function SWAP(a, b) that swaps the values of a and b.

(b) The algorithm MERGESORT operates recursively: unless the array A contains only one element, it is divided into two subarrays of (almost) the same size. These subarrays are ordered recursively and then merged into one sorted array.

MERGESORT(A)
Input: an array of integers $A[1 \ldots n]$

if $n = 1$ **then return** A
$m := \lceil n/2 \rceil$
array $A_1[1 \ldots m]$, $A_2[1 \ldots n-m]$ of **integers**
for $i := 1$ **to** m **do** $A_1[i] := A[i]$
for $i := 1$ **to** $n-m$ **do** $A_2[i] := A[i+m]$
$A_1 := \text{MERGESORT}(A_1)$
$A_2 := \text{MERGESORT}(A_2)$
$A := \text{MERGE}(A_1, A_2)$

Implement (in pseudo-code) the function MERGE(A_1, A_2) for merging two sorted subarrays A_1 and A_2 into one sorted array.

(c) The algorithm COUNTSORT creates an auxiliary array *count* and uses an entry *count*[i] to count how many elements in A have the value i.

COUNTSORT(A)
Input: an array of integers $A[1 \ldots n]$
$A_{\max} := \text{MAX}(A)$
$A_{\min} := \text{MIN}(A)$
array $count[A_{\min} \ldots A_{\max}] = [0, 0, \ldots, 0]$ of **integers**
for $i := 1$ **to** n **do** $count[A[i]] := count[A[i]] + 1$
$i := 1$
for $j := A_{\min}$ **to** $A_{\max}$ **do**
  for $k := 1$ **to** $count[j]$ **do** $\{ A[i] := j;\ i := i + 1; \}$

Implement (in pseudo-code) the functions MAX(A) and MIN(A) returning, respectively, the maximum and the minimum elements in the array A.

(d) The algorithm RADIXSORT may be viewed as a consecutive application of the algorithm COUNTSORT to the bits of the numbers. We start with the most significant bit of the integers in A (without loss of generality, they all have the same bit-length) and divide A into two arrays, according to the values of the most significant bit (0 or 1). For each of the arrays, containing more than 1 integer, we then look at the following bit, and divide the array into two, according to the value of that bit. The process is repeated until the least significant bit is reached. If some of the arrays appear to be empty, we immediately remove them from consideration. In particular, when the algorithm terminates, there are at most n arrays, each containing integers of the same value.

Define appropriate data structures and implement (in pseudo-code) the algorithm RADIXSORT(A) as described above.

For each of the described algorithms, compute its worst-case execution time and worst-case memory consumption. In particular, specify which of the algorithms are (1) strongly polynomial, (2) polynomial, and (3) pseudo-polynomial.