EXERCISES FOR CSE202

Exercise 1. *In the problem k-sum, you are given an n-tuple of nonnegative integers $L = (x_1, \ldots, x_n) \in \mathbb{N}^n$ and a target sum $S \in \mathbb{N}$. The problem is to determine whether there exists a subset of A of $\{1, \ldots, n\}$* of cardinality k *such that*

$$\sum_{i \in A} x_i = S.$$

1. *Show that this problem has a solution in polynomial complexity.*
2. *For $k = 2$ give an algorithm (in pseudo-code) solving the problem in $O(n \log n)$ operations (addition and comparison of integers are assumed to have constant cost).*
3. *Deduce an algorithm in $O(n^2)$ operations for $k = 3$ (give its pseudo-code).*

Solution : (1) For fixed $k \geq 1$, the following pseudo-code runs in time $O(n^k)$:

```
for $i_1$ in range(n):
 for $i_2$ in range($i_1 + 1$,n):
   ...
      for $i_k$ in range($i_{k-1} + 1$,n):
        if $L[i_1] + \cdots + L[i_k] == S$: return True
        else: return False
```

(2) We first sort L, which costs $O(n \log(n))$. Then for each position $j \in [1..n-1]$ we let ℓ_j be the smallest $i < j$ such that $L[i] + L[j] \geq S$. If no such i exists we let $\ell_j = j$. We note that there is a solution (i, j) (with $i < j$) to the 2-sum problem iff (ℓ_j, j) is a solution. Hence it suffices to run j from $n-1$ to 1, each time maintaining $\ell = \ell_j$ and testing if (ℓ_j, j) is a solution (if at some point we get $\ell = j$ then it means that $x[j-1] + x[j] < S$ hence there can be no solution (i, j') with $i < j' \leq j$ and thus we can return False in that situation). This is done by the following method

```
SUM2($L, S$):
  $j = n - 1$; $\ell = 0$
  while $j > 0$:
    while $\ell < j$ and $L[\ell] + L[j] < S$ :  $\ell = \ell + 1$
    if $\ell == j$: return False
    if $L[\ell] + L[j] == S$: return True
    $j = j - 1$
  return False
```

The complexity (assuming L is already sorted) is clearly $O(n)$, since the number of outer loop iterations is at most n, and the total number of inner loop iterations (summed over all iterations in j) is at most n since each such iteration lets ℓ increase by 1 (and ℓ can not get beyond $n - 1$). Hence the total cost for deciding if 2-sum has a solution is $O(n \log(n))$ (dominated by the cost of sorting).

(3) Again the first step is to sort L, which costs $O(n\log(n))$. Then we call the following method

```
SUM3($L, S$):
  for $i$ in range(n):
    $Lp = L[:i] + L[i+1:]; Sp = S - L[i]$
    if SUM2($Lp, Sp$): return True
  return False
```

For each i the cost is $O(n)$, hence the total cost is $O(n^2)$. Hence the total cost for deciding if 3-sum has a solution is $O(n\log(n)) + O(n^2) = O(n^2)$. □

Exercise 2. *Let* $\mathrm{Mul}(n)$ *denote the complexity of multiplying polynomials of degree at most* n *in* $\mathbb{K}[X]$ *in terms of number of arithmetic operations on the coefficients in the field* $\mathbb{K}$*, assumed to satisfy* $\mathrm{Mul}(n_1) + \mathrm{Mul}(n_2) \leq \mathrm{Mul}(n_1 + n_2)$ *and* $\mathrm{Mul}(mn) \leq m^2 \mathrm{Mul}(n)$*.*

For simplicity, n *is assumed to be a power of 2.*

(1) *Given* $(a_1, \ldots, a_n) \in \mathbb{K}^n$*, show that the coefficients of the polynomial* $(X - a_1)\cdots(X - a_n)$ *can be obtained in* $O(\mathrm{Mul}(n)\log n)$ *operations.*
(2) *Given* $P \in \mathbb{K}[X]$ *of degree* n *show that the values of* $P(a_1), \ldots, P(a_{n/2})$ *can be obtained from the values taken at* $(a_1, \ldots, a_{n/2})$ *by the polynomial* $R(X)$ *remainder of the Euclidean division of* P *by* $(X - a_1)\cdots(X - a_{n/2})$*.*
(3) *Deduce a divide-and-conquer algorithm for the evaluation of* P *at* $(a_1, \ldots, a_n)$ *(give pseudo-code).*
(4) *Estimate the complexity of that algorithm.*

Solution : (1) Assuming we have a method mult_poly that runs in time $O(\mathrm{Mul}(n))$, the following pseudo-code computes $(X - a_1)\cdots(X - a_n)$:

```
poly($a_1, ..., a_n$):
  if n==1: return $X - a_1$
  else: return mult_poly(poly($a_1, \ldots, a_{n/2}$), poly($a_{n/2+1}, \ldots, a_n$))
```

The running time $T(n)$ satisfies

$$\begin{aligned} T(n) &= \mathrm{Mul}(n/2) + 2T(n/2) \\ &= \mathrm{Mul}(n/2) + 2\,\mathrm{Mul}(n/4) + \cdots + 2^{k-1}\,\mathrm{Mul}(1), \text{ with } 2^k = n. \end{aligned}$$

Since $\mathrm{Mul}(n_1 + n_2) \geq \mathrm{Mul}(n_1) + \mathrm{Mul}(n_2)$, we have $\mathrm{Mul}(n) \geq m\,\mathrm{Mul}(n/m)$ whenever n/m is an integer. Hence $T(n) \leq k\,\mathrm{Mul}(n/2) = \log_2(n)\,\mathrm{Mul}(n/2) = O(\mathrm{Mul}(n)\log(n))$.

For the subsequent questions it is convenient to have a method tree$(a_1, \ldots, a_n)$ that outputs not only $(X - a_1)\cdots(X - a_n)$, but the full binary tree T whose root-node contains $(X - a_1)\cdots(X - a_n)$, left child of the root-node contains $(X - a_1)\cdots(X - a_{n/2})$, right child of the root-node contains $(X - a_{n/2+1})\cdots(X - a_n)$, etc. This method is

```
tree($a_1, ..., a_n$):
  if n==1: return BinaryTree($X - a_1$, None, None)
  else:
    left_subtree = tree($a_1, \dots, a_{n/2}$)
    right_subtree = tree($a_{n/2+1}, \dots, a_n$)
    root_poly = mult_poly(left_subtree.content, right_subtree.content)
    return BinaryTree(root_poly, left_subtree, right_subtree)
```

By the same analysis, this method has cost $O(\mathrm{Mul}(n)\log(n))$.

(2) If $P = (X - a_1)\cdots(X - a_{n/2})Q + R$ then clearly $P(a_i) = R(a_i)$ for $i \in [1..n/2]$.

(3) The methods are (using an auxiliary recursive method, and assuming we have already a method div(P,Q) that returns the remainder of the division of P by Q):

```
evalPoly($P, a_1, ..., a_n$):
  T=tree($P, a_1, ..., a_n$)
  return evalPolyAux($P, T, a_1, ..., a_n$)
```

```
evalPolyAux($P, T, a_1, ..., a_n$):
  if n==1: return [$P$]
  R1=div(P,T.left.content)
  R2=div(P,T.right.content)
  return evalPolyAux(R1,T.left,$a_1, ..., a_{n/2}$)
         +evalPolyAux(R2,T.right,$a_{n/2+1}, ..., a_n$)
```

The complexity $M(n)$ satisfies

$$M(n) = 2\mathrm{Div}(n/2) + 2M(n/2) = O(\mathrm{Mul}(n)) + 2M(n/2),$$

so that $M(n) = O(\mathrm{Mul}(n)\log(n))$ by the same analysis as in Question 1. □

Exercise 3. *If comparison is cheap, but writing is expensive, then the number of* exchanges *performed by Quicksort is an important parameter in the study of the time complexity. Assume that the algorithm is applied to a permutation of* $\{1, \dots, n\}$ *drawn uniformly at random.*

(1) *Show that the average number of exchanges during the first partition stage is* $(n-2)/6$.
(2) *Find the asymptotic behaviour of the average number of exchanges during the algorithm, first writing a linear recurrence that it satisfies.*

Solution : (1) Let L be the list to sort. The formula does not count the final swap involving the pivot (which always occurs unless the pivot is the smallest element), so we assume we do not count these swaps. Let $k = L[0]$ be the first element, used as the pivot. Let $i \in [1..k-1]$. The crucial property is that the position i is involved in a swap during the partition process iff $L[i] > k$. This occurs with probability $\frac{n-k}{n-1}$. Hence, conditioned on $L[0] = k$, the expected number of swaps involving i is $\frac{n-k}{n-1}$. Hence, conditioned on $L[0] = k$, the expected number of swaps is $(k-1)\frac{n-k}{n-1}$. Hence the expected number E_n of swaps is given by

$$E_n = \sum_{k=1}^{n} \frac{1}{n}(k-1)\frac{n-k}{n-1} = \frac{1}{n(n-1)}\sum_{k=1}^{n}(k-1)(n-k).$$

Here are two possible ways of computing this sum:

1. Let $S(n,p)=\sum_{k=1}^{n}\binom{k}{p}$. We have the formula $S(n,p)=\binom{n+1}{p+1}$ (proof by induction, note that $S(n,p)$ satisfies the recurrence $S(n,p)=\binom{n}{p}+S(n-1,p)$, and $T(n,p):=\binom{n+1}{p+1}$ also satisfies this recurrence, which is Pascal's triangle identity $\binom{n+1}{p+1}=\binom{n}{p}+\binom{n}{p+1}$). Then we have

$$\begin{aligned}E_n &= \frac{1}{n(n-1)}(nS(n-1,1)-2S(n,2))\\ &= \frac{1}{n(n-1)}(n\binom{n}{2}-2\binom{n+1}{3})=\frac{n-2}{6}\end{aligned}$$

2. $(k-1)(n-k)=-k^2+(n+1)k-n$ is to be summed with respect to k. The sum is obtained by linear combination from simple sums:

$$\sum_{k=1}^{n}1=n,\quad \sum_{k=1}^{n}k=\frac{n(n+1)}{2},\quad \sum_{k=1}^{n}k^2=\frac{n(n+1)(2n+1)}{6}.$$

In order to find the last one, one way is to observe that it has to be a polynomial of degree 3 and compute

$$(k+1)^3-k^3=3k^2+3k+1,$$

so that

$$\sum_{k=1}^{n}k^2=\frac{(n+1)^3-1}{3}-\frac{1}{3}\sum_{k=1}^{n}(3k+1)=\frac{n^3+3n^2+3n}{3}-\frac{n(n+1)}{2}-\frac{n}{3}=\frac{2n^3+3n^2+n}{6}.$$

(2) The calculation is very similar to that of the number of comparisons. The expected number C_n of swaps satisfies

$$C_n=\frac{n-2}{6}+\frac{1}{n}\sum_{k=1}^{n}C_{k-1}+C_{n-k}=\frac{n-2}{6}+\frac{2}{n}S_{n-1},\quad \text{with } S_n:=\sum_{k=0}^{n}C_k,$$

so we have $nC_n-n\frac{n-2}{6}=2S_{n-1}$, which gives $nC_n-(n-1)C_{n-1}-\frac{n(n-2)}{6}+\frac{(n-1)(n-3)}{6}=2C_{n-1}$, which is $nC_n-(n+1)C_{n-1}=\frac{2n-3}{6}$, i.e., we have

$$\frac{C_n}{n+1}-\frac{C_{n-1}}{n}=\frac{1}{n(n+1)}\frac{2n-3}{6}=\frac{5}{6(n+1)}-\frac{1}{2n}.$$

This gives

$$\frac{C_n}{n+1}=\sum_{k=1}^{n}\frac{5}{6(k+1)}-\frac{1}{2k}=-\frac{5}{6}+\frac{5}{6(n+1)}+\frac{1}{3}H_n,\quad \text{with } H_n=\sum_{k=1}^{n}\frac{1}{k}\sim\log(n).$$

To conclude we find $C_n\sim\frac{1}{3}n\log(n)$. □

Exercise 4. *In the Multiset Identity Problem, you are given two multiset of integers (i.e., sets with repetitions) and the problem is to decide whether they are equal or not.*

(1) *Give a first solution in $O(n\log n)$ operations.*
(2) *Find a better algorithm using hashing.*

Solution : (1) We have the method

```
equal(L1,L2):
  L1.sort(); L2.sort()
  return L1==L2
```

This method runs in time $O(n \log(n))$.

(2) We can then do faster by filling in dictionaries (keys are elements in the list, the associated value gives the number of times the key appears).

```
equal_faster(L1,L2):
  dict1={}
  for x in L1:
    if not x in dict1: dict1[$x$] = 1
    else: dict1[$x$] = dict1[$x$] + 1
  dict2={}
  for x in L2:
    if not x in dict2: dict2[$x$] = 1
    else: dict2[$x$] = dict2[$x$] + 1
  return dict1==dict2
```

Assuming cost $O(1)$ to access a key in a dictionary and update the associated value, this method has cost $O(n)$.

Here is a variant using only one dictionary

```
equal_faster(L1,L2):
  if len(L1)!=len(L2): return False
  dict={}
  for x in L1:
    if not x in dict: dict[$x$] = 1
    else: dict[$x$] = dict[$x$] + 1
  for x in L2:
    if not x in dict or dict[x]==0: return False
    else: dict[$x$] = dict[$x$] − 1
  return True
```

□

Exercise 5. *Construct a worst-case example for the Boyer-Moore algorithm over a binary alphabet in the simple variant where only the last character shift heuristic is used (ie, the max in the loop does not use the array bms).*

Solution : With the text a^{2k} and the pattern ba^{k-1}, the largest possible number of characters (k) is compared before the smallest possible shift (1) is performed. The complexity is quadratic. □

Exercise 6. *Prove that the following decision problem is NP-complete. Given n knights, and a set of pairs of knights who are enemies, is it possible to arrange the knights around a round table so that two enemies do not sit side by side? [Find a reduction involving one of the NP-complete problems seen in the course.]*

Solution : We construct a graph G as follows: the vertices are the knights, and for each pair of knights K, K', there is an edge between K and K' iff K and K' are not enemies. Then clearly, finding a table configuration is equivalent to finding a Hamiltonian cycle in G (the positions of the knights around the tables corresponding to their cyclic order around a given Hamiltonian cycle). Hence this problem is the same as finding a Hamiltonian cycle in a graph, which is NP-complete. □

Exercise 7. *The decision problem SET-COVER is defined as follows: Given a set X containing n elements, m subsets $S_1, \dots, S_m$ of X and an integer $k \le n$, is there a set of k of the subsets that covers all the elements of X? In other words, is there a subset I of $\{1, \dots, m\}$, of cardinality k, such that for all $x \in X$, $x \in \cup_{i \in I} S_i$?*

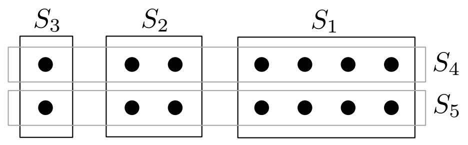
SET-COVER is an NP-complete problem. The associated optimization problem is to find the smallest cover. We consider the following greedy algorithm for this optimization problem:

- *Pick a subset S_j that covers the largest number of elements of X;*
- *Suppress from X and from the remaining S_is the elements covered by S_j, and start again.*

Analyse this algorithm by answering the following questions:

(1) *Is this algorithm optimal?*
(2) *Prove that if an optimal solution contains k subsets, the algorithm takes at most $O(k \log n)$ subsets.*
(3) *What is the approximation ratio of this algorithm?*

Solution : (1) It is not optimal, for instance in the following example:



The greedy algorithm first chooses S_1, then S_2, then S_3, so it gives a set-cover of cardinality 3, whereas the optimal solution is to take S_4, S_5, of cardinality 2.

(2) Since the optimal solution uses k sets, there must some set that covers at least a $1/k$ fraction of the elements. The algorithm chooses the set that covers the most elements, so it covers at least that many. Therefore, after the first iteration of the algorithm, there are at most $n(1 - 1/k)$ elements left. Again, since the optimal solution uses k sets, there must be some set that covers at least a $1/k$ fraction of the remaining elements. So, again, since we choose the set that covers the most elements remaining, after the second iteration, there are at most $n(1 - 1/k)^2$ elements left. More generally, after t rounds, there are at most $n(1 - 1/k)^t$ elements left. After $t = k \log(n)$ rounds, there are at most $n(1-1/k)^{k \log(n)} < n(1/e)^{\log(n)} = 1$ elements left, which means we must be done.

(3) The previous question shows that the approximation ratio is at most $O(\log n)$. The example of Question 1 can be generalized to show that indeed this bound is reached, so that the worst-case ratio for n elements is $\Theta(\log(n))$. Take $n = 2 + 4 + 8 + \cdots + 2^{k+1} = 2^{k+2} - 2$ elements, a set S_1 of cardinality 2^k, S_2 of cardinality 2^{k-1},…, S_k of cardinality 2, and two further sets S_{k+1} and S_{k+2} each of cardinality $2^k - 1$ (the figure above shows the case $k = 3$). The greedy algorithm will pick S_1, then S_2,…, then S_k, yielding a set cover of cardinality $k \sim \log n$,

whereas the optimal set-cover, obtained by picking S_{k+1} and S_{k+2}, has constant cardinality 2. □