Analysis: ELE
Elections

HISTORY:
• v. 1.01: 01.04.2008, WTYC – small corrections
• v. 1.00: 23.03.2008, Mi.Pi. — writing the analysis

dokument systemu SINOL 1.7.2

1 Introduction

This task is a simple variation of discreet backpack problem. Instead of items packed in a backpack we have parties with certain numbers of seats in a parliament to form a coalition. In this problem we do not have to find out, if a given amount of votes can be obtained, but what is the maximal size of a possible, non-redundant coalition. Non-redundancy makes the problem a little bit harder — during dynamic algorithm one has to somehow ensure that constructed coalition will not be redundant, which can occur when trying to apply standard methods.

The solution of this problem is considering the parties in non-ascending order of number of seats. It is a simple observation that if one breaks the minority when adding a party to a coalition consisting only of larger parties, the new coalition will be proper and non-redundant. Therefore, after sorting the sizes of parties we use the standard dynamic algorithm to consider subsets of parties which do not have majority and record only 'skips' over the half.

This 'trick' makes the problem more interesting, but still it is rather standard and easy.

2 Jury's solution

Firstly, let us observe that a coalition is non-redundant if without a member party with the smallest number of seats it hasn't got the majority in the parliament. Obviously every non-redundant coalition has this property. Moreover, if exclusion of the smallest party breaks the majority, then exclusion of each larger will break as well, so each coalition with this property will be indeed non-redundant. Going further, all non-redundant coalitions are created from subsets not having majority by adding a party not larger than any already included.

In the first step of the algorithm the results of parties are to be sorted in non-ascending order of the number of seats. As there is at most 300 parties, we can do this using sorting algorithms with complexity $O(n^2)$, or use template ones in C/C++ instead. Now we can assume that if $a_1, a_2, \ldots, a_n$ are the number of seats of parties then $a_1 \geq a_2 \geq \ldots \geq a_n$.

In the second step we apply dynamic programming approach. Let s be the total number of seats in the parliament, computed during input reading. We will use two arrays with size `[0..s]`: `partyUsed[i]` and `partySkip[i]`. Let us assume that we have already considered the first k largest parties. In `partyUsed[i]` we will store the number of the smallest party of a subset of considered parties, which has exactly i seats in total. If such a subset does not exist, -1 will be stored. In `partySkip[i]` the size of party from `partyUsed[i]` shall be recorded. It will be useful further to reconstruct the optimal coalition. The information in those two arrays will be stored only for subsets of parties not having majority and for non-redundant coalitions.

It is easy to observe that such a data structure can be maintained during considering the parties one after another. Firstly, `partyUsed` is filled with -1. If we consider k-th party, we add its number of seats to all values not exceeding $\lfloor \frac{s}{2} \rfloor$, obtained in previous steps. Thus we construct new subsets of parties, each including this new party. We record all the new possible numbers of seats built in this way, changing appropriate value in `partyUsed` to k and in `partySkip`to corresponding number of votes. The values having majority are being

recorded in this step, but they are not included in the set of values, to which we add the considered number of seats. The observation in the first paragraph ensures us that all such number of seats will be results of non-redundant coalitions, as the parties were considered in non-ascending order of number of seats.

In order to quickly find out, which numbers of seats have been obtained so far, we store them in an additional container — for example a vector or an array. If a new number of seats below majority has been constructed, we simply add information about it to this structure. In the beginning, only 0 is included.

In the third step we are to reconstruct the optimal coalition. Firstly we seek its number of seats. We can do it by finding the largest index with nonnegative value in `partyUsed`, or by remembering it during performing the previous step. Then we are able to reconstruct the answer. We begin with a variable *best* set to the found index and decrement it by `partySkip[best]` while recording usage of `partyUsed[best]`. After reaching 0, the whole coalition has been recorded and is ready to be printed. This algorithm is correct, because parties with no seats will never be used, as they are always redundant.

The first step of the algorithm, implemented efficiently, has time complexity $O(n \log n)$. The second one takes $O(n \cdot s)$ time — for each of n parties we consider at most $\lfloor \frac{s}{2} \rfloor$ recorded values. The last step has complexity $O(n)$, as the answer cannot be larger. For reasonable data set we have $s \geq n$, so the time complexity of the whole algorithm is $O(n \log n + n \cdot s) = O(n \cdot s)$. It is noteworthy that the usage of inefficient sorting algorithms does not worsen the time complexity.

The memory complexity is clearly $O(s)$ because of usage of arrays of this size in dynamic programming.

The implementation of this algorithm is included in files:

- `ele.cpp` — implementation in C++,
- `ele1.c` — implementation in C,
- `ele2.pas`— implementation in PASCAL.

3 Other solutions

3.1 Efficient solutions

Algorithm 1. *(100 points)* This is in fact a bit less efficient implementation of jury's solution. We do not use a container to store visited states — instead, during considering a single party, the whole array `partyUsed[]` is read to find sets of parties with new numbers of seats. The time and memory complexities are obviously the same, but this implementation runs a bit slower due to senseless processing of the whole `partyUsed[]` in early states of the algorithm.

The implementation can be found in files `ele3.cpp` (C++) and `ele4.pas` (PASCAL).

3.2 Inefficient solutions

Algorithm 1. *(40 points)* The first inefficient algorithm is brutal backtracking with complexity $O(n \cdot 2^n)$. For every subset of parties (2^n of them), we check if a coalition can be formed out of them and if it is non-redundant (which takes us $O(n)$ time). The solution is straightforward and scores only guaranteed set of points.

The implementation can be found in files `eles1.cpp` (C++) and `eles2.pas` (PASCAL).

Algorithm 2. *(50 points)* The algorithm is a brief variation of the first algorithm. While recurrently constructing subsets, we compute their number of votes and check only those having majority. Moreover, after achieving more than a half of all the seats, we do not need to continue the search, as every further coalition will be redundant. We also cut the search if we are sure that even including all the remaining parties will not

result in having the desired majority. These implementation tricks make the algorithm a bit faster, but it is still exponential. However, it passes two more tests.

The implementation can be found in files `eles3.cpp` (C++) and `eles4.pas` (PASCAL).

Algorithm 3. *(75 points)* This algorithm attempts to apply standard dynamic programming, but in a inefficient way. Firstly, we observe that a coalition will be non-redundant if excluding the smallest party breaks its majority. In the algorithm, for every party we assume it to be this smallest one. From larger ones we try to obtain a subset with the biggest possible number of seats, but not exceeding the half of all the seats. If we add the assumed minimal party, we will form a maximal non-redundant coalition with the chosen party as the smallest one. In order to do this, we use standard dynamic programming approach using arrays `partySkip[]` and `partyUsed[]`, but considering only parties with at least so many seats as the assumed minimal one. In addition, we perform the search only if the total number of seats claimed by those parties, including the assumed one, is more than the half of the parliament. Only if this condition is satisfied, a coalition can be formed. The maximal coalition is the best one found in any of the described steps.

The time complexity of this algorithm is $O(n^2 \cdot s)$ — for each party we run the whole dynamic algorithm with complexity $O(n \cdot s)$.

The implementation can be found in files `eles5.cpp` (C++) and `eles6.pas` (PASCAL).

3.3 Incorrect solutions

Algorithm 1. *(0 points)* The first heuristic algorithm sorts the parties in descending order of number of seats. Then, starting from every party it tries to grab the biggest possible parties until a coalition can be formed. The formed coalition will surely be non-redundant from the same reasons as in the jury's solution. The algorithm runs in $O(n^2)$ time, but it assumes that the optimal coalition is formed by a sequence of consecutive parties in order of number of seats. This assumption is obviously not true.

The implementation can be found in file `eleb1.cpp` (C++).

Algorithm 2. *(35 points)* This is a randomized version of the first inefficient algorithm. It randomly chooses many subsets of parties and finds out, which of them form a non-redundant coalition. The best considered coalition is being outputted. For small tests the algorithm runs much like the deterministic version, as there is not much subsets to consider and the chance of finding the best one as well is quite big. However, for bigger tests the algorithm doesn't find even a subset able to form a non-redundant coalition, not to mention the best one.

The implementation can be found in file `eleb2.cpp` (C++).

Algorithm 3. *(0 points)* This algorithm tries to solve problems in the first incorrect algorithm. Once more the parties are sorted, and we begin a procedure of grabbing biggest parties starting from each party. If inserting a new party into a set of already grabbed doesn't break the minority, we grab it. Otherwise we try to grab it, see if a new non-redundant coalition is larger than previously one found and store it if needed. Then we continue the algorithm as if we did not grab the considered party.

The algorithm runs in $O(n^2)$ time and checks a bit more coalitions than the first algorithm. However, it is still incorrect.

The implementation can be found in file `eleb3.cpp` (C++).

Algorithm 4. *(0 points)* This algorithm is an attempt to apply a standard dynamic programming approach not without previously sorting the parties. As a result, formed coalitions are merely redundant.

The implementation can be found in file `eleb4.cpp` (C++).

4 Constraints

The constraint of the number of parties ($1 \leq n \leq 300$) is left unchanged. Instead of introducing a bound for a number of seats of each party, the whole number of seats has been bounded by 100 000. The combination of those two constraints make it easy to differ efficient solutions from inefficient ones, while not bounding result of each party leaves much more flexibility in testing contestants' solutions. Furthermore, no changes have to be applied to the jury's solution, as its time complexity depend only on these two bounded variables. It is noteworthy that a lower bound for the number of seats has been introduced — it has to be positive, because for zero number of seats the problem has no sense.

The memory limit has been set to standard $32Mb$ — the memory matters are not important in this task. The time limit has been set to $1s$. — inefficient solutions are significantly exceeding it, while for efficient ones there is no problem in fitting into.

It is guaranteed that correct but inefficient solutions, performing well for not above 20 parties, will score at least 40% of points. The constraint has been chosen in order to give even the most brutal, exponential back-tracking algorithms a number of points for correctness.

5 Tests

The following tests have been prepared (s is the total number of seats in the parliament):

- `ele0.IN` (1 sek.) sample test from the problem statement, $n = 4, s = 10$
- `ele1ocen.IN` (1 sek.) simple small test, $n = 12, s = 2048$
- `ele2ocen.IN` (1 sek.) simple small test, $n = 8, s = 28$
- `ele3ocen.IN` (1 sek.) simple small test, $n = 5, s = 24$
- `ele4ocen.IN` (1 sek.) simple large test, $n = 300, s = 45150$
- `ele1.IN` (1 sek.) small test, $n = 6, s = 156$
- `ele2.IN` (1 sek.) small test, $n = 11, s = 43$
- `ele3.IN` (1 sek.) small test, $n = 12, s = 79$
- `ele4.IN` (1 sek.) small test, $n = 13, s = 3272$
- `ele5.IN` (1 sek.) small test, $n = 14, s = 11592$
- `ele6.IN` (1 sek.) small test, $n = 16, s = 75981$
- `ele7.IN` (1 sek.) small test, $n = 18, s = 86130$
- `ele8.IN` (1 sek.) small test, $n = 20, s = 52361$
- `ele9.IN` (1 sek.) small test, $n = 26, s = 22041$
- `ele10.IN` (1 sek.) small test, $n = 27, s = 37395$
- `ele11.IN` (1 sek.) medium test, $n = 45, s = 97154$
- `ele12.IN` (1 sek.) medium test, $n = 48, s = 99588$

- `ele13.IN` (1 sek.) medium test, $n = 72, s = 49306$
- `ele14.IN` (1 sek.) medium test, $n = 110, s = 15936$
- `ele15.IN` (1 sek.) medium test, $n = 100, s = 47236$
- `ele16.IN` (1 sek.) large test, $n = 250, s = 84948$
- `ele17.IN` (1 sek.) large test, $n = 270, s = 99979$
- `ele18.IN` (1 sek.) large test, $n = 300, s = 72267$
- `ele19.IN` (1 sek.) large test, $n = 300, s = 79669$
- `ele20.IN` (1 sek.) large test, $n = 300, s = 99945$

Most of the tests are based on random choice of results of the parties from a certain interval. This method of test data generation appears to be quite effective against heuristic incorrect algorithms as well as inefficient ones.

The contestants are provided with tests $[1-4]$*ocen*. They are very simple examples and, traditionally, all incorrect solutions perform on them quite well.

The main test set are tests $1-20$. Each of them is worth 5 points.

Tests $1-8$ have $n \leq 20$ and are worth exactly 40% of points. Tests $1-10$ can be passed by solutions with exponential complexity, however to fit into the limit in tests 9 and 10, the algorithm has to be a bit more sophisticated. Tests $11-15$ represent medium size of test data and can be solved by algorithms with complexity a bit worse than jury's. Tests $16-20$ are the largest ones, passable only by efficient solutions.

6 Changes in the problem statement

A constraint concerning the number of seats of each party has been replaced by a sentence 'You may assume that the total number of seats in the parliament will be positive and lower or equal to 100 000.'.

A sentence concerning guaranteed points for inefficient solutions has been added to the Input section.

Reasons for those changes are discussed in section 'Constraints'.

7 Remarks

The task is rather easy. It is a standard example of dynamic programming approach. A few 'tricks' are used inside, but still there is nothing 'new' in the general manner. In spite of this, the task can be regarded as an interesting proposition for an easy or medium problem for the contest.

The time limits have to be adjusted to the system on which the contestants' solutions will be judged. The 1 second limit is recommended, as on my computer it differs inefficient solutions from efficient ones, but on other architecture the performance of the algorithms might be slightly slower or faster. The third inefficient algorithm should solve tests $16-20$ at least $2-3$ times longer than the limit.

8 List of work done

- Creation of this document,
- creation of the test set,

- 3 implementations of efficient algorithms in C/C++ and 2 in PASCAL,
- 3 implementations of inefficient algorithms in C++ and 3 in PASCAL,
- 4 implementations of incorrect algorithms in C++,
- implementation of a program verifying the correctness of input data,
- implementation of a program judging the contestant's solutoin,
- testing of the programs and setting the limits.