Starting from this week, you will be developing a computer program to investigate the practical computational complexity of the *Subset Sum Problem (SSP)* or the *Partition Problem (PP)*, for your Coursework 2.

This guide will be updated at least weekly, so make sure you always uses the latest version (see the right footer).

Since no model answers can be made available, Kamal will instead investigate the **Satisfiability problem (SAT)** – watch
https://github.com/ab3735/380CT/tree/master/SAT

Consider the following two problems

Subset-Sum Problem (SSP)

Given a set $S = \{x_1, x_2, \ldots, x_n\}$ of integers, and an integer t (called *target*) decide if there is a subset of S whose sum is equal to t.

Partition Problem (PP)

Given a set $S = \{x_1, x_2, \ldots, x_n\}$ of numbers, decide if it can be partitioned into two sets such that they both have the same sums.

First note that a PP instance can easily be converted into an SSP instance as follows:

Given a set $S = \{x_1, \ldots, x_n\}$ for PP, calculate $t = (x_1 + \cdots + x_n)/2$.

- If t is odd then this PP instance has a "No" answer. (Why?)
- Otherwise, create an SSP instance with the same set S and target t as just calculated.

The discussions hereafter will be specific to SSP using Python, but can easily be adapted to PP (using the reduction sketched above) and other programming languages. GitHub or similar may be used.

(1) Create a GitHub repository for this project.

Keep it **private** to start with. Make it public after the submission deadline.

If you are working as a group then make sure every member is invited and has access, and that you contribute to the work fairly between you.

(2) Download the basic Python code from Moodle: `SSP.py`. Open it using e.g. IDLE.

Please note that this is written for Python 3.

(3) Read the code. What are the classes attributes and methods? What are they meant to do? How are they used?

(4) Run it. What does it do? Can you make it try SSP instances of different sizes?

(5) *Explore* the code and see what you can do with it.

(6) Improve it! Start by adding the missing docstrings.

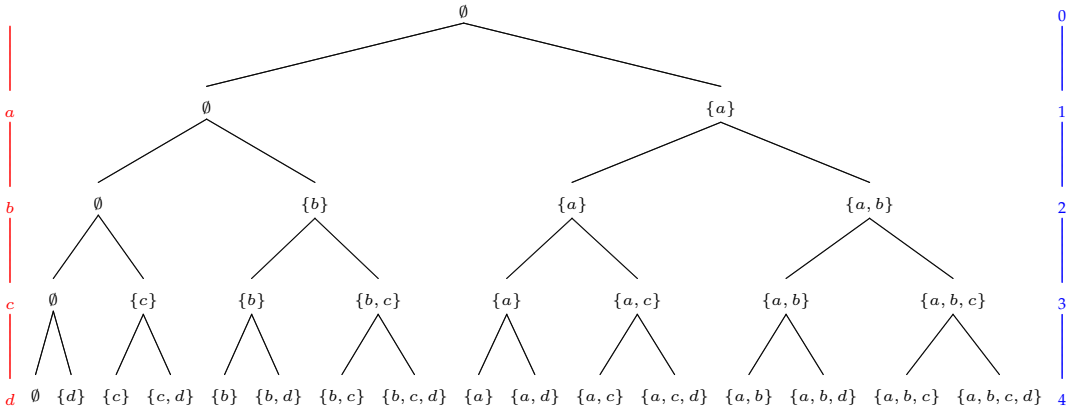
(1) On paper, think about how to exhaustively search for the solution to a given SSP instance. You may write pseudo-code, or just outline your method.

Express the complexity of your method using O-notation.

Hint: Enumerate all the subsets of S. (Exponential running time).

You can think of your exhaustive search as a binary tree, where at each level you decide whether to include the i-th item from S or not.

For example, if $S = \{a, b, c, d\}$ then you get



(2) Implement a method to try and find a solution using exhaustive search.

This should return `True` if a solution exists, and `False` otherwise.

(3) Experiment with the time it takes to solve random instances of sizes $n = 30, 31, 32, \ldots$ until it starts taking "too long".

Can you obtain "average case" and "worst case" results?

What do we get for the "best case" scenario?

(4) Plot the times obtained from the previous question as a function of n, and try to extrapolate the time needed for the values of n up to 100.

(5) Try to optimize your code using the optimization options available to your programming language.

How much difference does this make?

In the last section, we looked at the subsets of S as being the **search space**. Another way to search for solutions is to build the answer for smaller target values $t' = 1, 2, 3, \ldots, t$. If t is small enough then this yields an efficient method even if n is large (unfeasible with exhaustively searching the space of subsets of S).

This idea is general and is called **Dynamic Programming**.

(1) If L is a set of integers and x is another integer, then we use the shorthand $L + x$ to denote the list of integers derived from L by increasing each element of L by x, i.e. $L + x = \{\ell_k + x \mid \ell_k \in L\}$.

For example, if $L = \{1, 5, 9\}$, then $L + 3 = \{4, 8, 12\}$.

1: $L \leftarrow \{0\}$
2: **for** $x \in \{x_1, \ldots, x_n\}$ **do**
3: $\quad L \leftarrow L_{\cup}(L + x)$
4: $\quad$ remove from L every element that is greater than t
5: **end for**
6: **return** True if $t \in L$, False otherwise

(2) Implement a dynamic programming search method.

Check http://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/ for sample code.

Note that you do not really need a 2D array, a 1D array is sufficient as shown in the pseudocode above.

(3) Experiment with the time it takes to solve random instances of increasing bit-length of t, and compare your results with those from exhaustive search.

When is it best to use one of these two methods?

(4) Experiment with the time it takes to solve random instances of sizes $n = 30, 31, 32, \ldots$ until it starts taking "too long" and compare the results to the times obtained from the last section.

We have so far considered methods that guarantee an exact answer. We shall now turn our attention to faster methods but at the expense of exactness.

A **greedy algorithm/method** tries to approximate the solution to an *optimization problem* by successively adding a component to an approximate solution. The choice of the component at each step is made to maximize the *immediate gain*, without considering the long-term effect.

Such methods are usually quite efficient in practice, and sometimes may be the only practical option with very large problem instances.

The *optimization* version of the **Subset Sum problem (SSP)** is defined as follows:

> Given a set $S = \{x_1, x_2, \ldots, x_n\}$ of positive integers and a positive target integer t, find a subset of S whose sum **minimizes** the distance to the target t?

(1) Consider the following greedy algorithm:

1: $total \leftarrow 0$
2: **while** there exists some $x_i \in S$, not previously selected, such that $total + x_i \leq t$ **do**
3: $\quad total \leftarrow total + x_i$ where x_i is the largest element of S such that $total + x_i \leq t$
4: **end while**
5: **return** $t - total$ (and the subset of selected x_i's)

This does not guarantee correctness but is very fast.

- Find examples where this method fails to find the correct answer.
- What is its time complexity in O-notation?

(2) Implement a greedy method following the outline above.

(3) Experiment with the time it takes to solve random instances, and compare your results with previous methods.

Once a greedy (approximate) solution is constructed we can perform a *local search* to try and improve the candidate solution. This can, for example, be done by swapping an x_i in the candidate solution with an x_j in S which is not in the solution candidate. If this results in an improvement then the new candidate is chosen, otherwise the swapping is undone, and another attempt is made.

(1) The **Greedy Randomized Adaptive Search Procedure (GRASP)** essentially employs the previous idea as follows:

```
1: best_candidate ← ∅
2: while (termination condition is not met) do
3:     greedy_candidate ← ConstructGreedyRandomizedSolution()
4:     grasp_candidate ← LocalSearch(greedy_candidate)
5:     if f(grasp_candidate) < f(best_candidate) then
6:         best_candidate ← grasp_candidate
7:     end if
8: end while
9: return best_candidate
```

Here:

- f is an *objective function* which measures how "good" a given candidate is.

  In the SSP case, this can be taken to be $t - total$ or $|t - total|$.

- ConstructGreedyRandomizedSolution() is a function that generates a solution candidate using a randomized greedy selection process.

- LocalSearch(s) performs a local search in the *neighbourhood* of s. This can be done as explained above using swapping of elements.

- "termination condition" could be a maximum number of iterations, etc.

You may want to refer to the **Handbook of metaheuristics** for detailed discussion (pp. 219–249) (http://site.ebrary.com/lib/coventry/docDetail.action?docID=10067312).

> "GRASP is a multi-start metaheuristic for combinatorial problems, in which each iteration consists basically of two phases: construction and local search. The construction phase builds a feasible solution, whose neighborhood is investigated until a local minimum is found during the local search phase. The best overall solution is kept as the result."

Try to implement this method for SSP.

(2) Experiment and compare to Greedy.

Below are pseudocodes for a few further meta-heuristics. Choose one or more and experiment!

(1) Iterative Improvement

determine initial candidate solution s
while s is not a local optimum **do**
    choose a neighbour s' of s such that $f(s') < f(s)$
    $s \leftarrow s'$
end while
return s

(2) Randomized Iterative Improvement

choose a probabilty threshold $w \in [0..1]$
determine initial candidate solution s
while termination condition is not satisfied **do**
    randomly generate a number $p \in [0..1]$
    if $p < w$ **then**
        choose a neighbour s' of s uniformly at random
    else
        choose a neighbour s' of s such that $f(s') < f(s)$
        or if no such s' exists then choose s' such that $f(s')$ is minimal
    end if
    $s \leftarrow s'$
end while
return s

(3) Simulated Annealing

determine initial candidate solution s
set initial temperature T according to annealing schedule
while termination condition not satisfied **do**
    probabilistically choose a neighbour s' of s
    if s' satisfies probabilistic acceptance criterion (depending on T) **then**
        $s \leftarrow s'$
    end if
    update T according to annealing schedule
end while
return s

(4) Tabu Search

determine initial candidate solution s
while termination condition not satisfied **do**
    determine set N of non-tabu neighbours of s
    choose a best improving solution s' in N
    update tabu attributes based on s' $s \leftarrow s'$
end while
return s

(5) Ant Colony Optimization

initialize weights
while termination criterion is not satisfied **do**
    generate population sp of candidate solutions using subsidiary randomized constructive search
    perform subsidiary local search on sp
    adapt weights based on sp
end while
return s

The subsidiary constructive search uses weights (pheromone trails) and heuristic information.

(6) Genetic Algorithm

1: determine initial population p
2: **while** termination criterion not satisfied **do**
3: generate set pr of new candidate by recombination
4: generate set pm of new candidates from p and pr by mutation
5: select new population p from candidates in p, pr, pm
6: **end while**

Implement a method to detect special cases that can be solved exactly in polynomial time.

Below are some cases for you to investigate.

(1) There is a trivial case where t is bigger than the total sum possible with S. What should the answer be then?

(2) What happens if t is smaller than the smallest element in S?

(3) Think about the following instance

$$S = \{1, 2, 3, 10, 20, 30, 100\}, \quad t = 13.$$

What are the elements of S that you can"ignore"? This should give you a smaller "effective" set $S' \subset S$; hence reducing the size of the problem, and consequently decreasing the cost of the computation needed.

(4) What happens if the elements of S have a common factor $d > 1$ which is not a factor of t?

(5) When is the greedy approach guaranteed to succeed in finding an exact solution?

How often do these special cases occur?