This practical session consists in implementing and analysing common scheduling strategies for the problem $P||C_{max}$. Use the language in which you are the most efficient.

The first two sections are essential.

1 Managing your Workspace

Launch `du -s -h` in a terminal to determine the size of your workspace. While this varies a lot, the connection time is roughly 2 to 5 minutes per Gigabyte. It is desirable to reduce the workspace to under 1 or 2 GB. You can launch `baobab`, the Gnome application named “Disk Usage Analyzer”, to have a visual aid for this.

The actual cleaning must be done outside this practical session.

2 Scheduling Algorithms Implementation

Write a function that generates an array of $n$ integer task execution times. The generation is assumed to be random uniformly between 1 and $R$ (for the tests, it is possible to set these parameters to $n = 10$ and $R = 10$ for instance).

In Python:

```python
import random
def gen_cost(n, M):
    return [random.randrange(1, M + 1) for _ in range(n)]
```

In Java:

```java
int[] genCost(int n, int M) {
    return DoubleStream.generate(Math::random)
        .limit(n)
        .mapToInt(x -> (int)(M * x) + 1)
        .toArray();
}
```

The maximum completion time of any schedule is bounded by 2 values: the sum of the execution times divided by the number of processors, and the maximum execution time of any task. Implement a function that computes the maximum between these two lower bounds from the execution times and the number of processors.
def LB(cost, m):
    return max(sum(cost) / m, max(cost))

double LB(int[] cost, int m) {
    return Math.max(Arrays.stream(cost).sum() / m, Arrays.stream(cost).max().orElse(0));
}

Implement LPT, MULTIFIT and SLACK. Each function takes as input the execution times and a number of processors and returns the maximum completion time of the schedule.

def LST(cost, m):
    C = [0] * m
    while cost:
        C[0] += cost.pop(0)
    C = sorted(C)
    return max(C)

def LPT(cost, m):
    return LST(sorted(cost, reverse=True), m)

def FFD(cost, m, mks):
    cost = sorted(cost)
    C = [0] * m
    while cost:
        i = 0
        curr = cost.pop()
        while i < m and C[i] + curr > mks:
            i += 1
        if i == m:
            return False
        C[i] += curr
    return True

import math
def MULTIFIT(cost, m):
    lb = math.ceil(LB(cost, m))
    if FFD(cost, m, lb):
        return lb
    up = math.ceil(max(lb, 2 * sum(cost) / m))
    while lb + 1 != up:
        mid = (lb + up) // 2
        if FFD(cost, m, mid):
            up = mid
        else:
            lb = mid
    return up

def SLACK(cost, m):
    cost = sorted(cost, reverse=True)
group = []
for i in range(0, len(cost), m):
group.append(cost[i:i + m])
group[-1] += [0 for _ in range(m - len(group[-1]))]
group = sorted(group, key = lambda x: x[0] - x[-1], reverse = True)
cost = []
for i in group:
    cost += i
return LST(cost, m)

def OPT(cost, m, C = None):
    if C is None:
        C = [0] * m
    if not cost:
        return max(C)
    mks = sum(cost) + max(C)
    for i in range(m):
        C[i] += cost[0]
        mks = min(mks, OPT(cost[1:], m, C))
        C[i] -= cost[0]
    return mks

int LST(int[] cost, int m) {
    var C = IntStream.generate(() -> 0).limit(m).toArray();
    Arrays.stream(cost)
        .forEach(t -> {
            Arrays.sort(C);
            C[0] += t;
        });
    return Arrays.stream(C).max().orElse(0);
}

int LPT(int[] cost, int m) {
    var C = Arrays.stream(cost)
        .boxed()
        .sorted(Comparator.reverseOrder())
        .reduce(IntStream.iterate(0, x -> 0)
            .limit(m).toArray(),
            (acc, t) -> {
                Arrays.sort(acc); acc[0] += t; return acc;
            }, (acc1, acc2) -> null);
    return Arrays.stream(C).max().orElse(0);
}

int SPT(int[] cost, int m) {
    var C = Arrays.stream(cost)
        .sorted()
        .collect(() -> IntStream.range(0, m)
            .map(i -> 0)
            .toArray(),
            (acc, t) -> { Arrays.sort(acc); acc[0] += t; },
            (acc1, acc2) -> {});
    return Arrays.stream(C).max().orElse(0);
}
3 Worst-case Instances

Find an instance with the minimum number of processors and tasks such that LPT outperforms both other heuristics. Idem for MULTIFIT and SLACK.

For LPT, this leads to the following task costs for \( m = 2 \) processors: \( \{ p_j \} = \{ 7, 3, 3, 3, 2, 2, 2 \} \). For MULTIFIT: \( \{ p_j \} = \{ 3, 3, 2, 2, 2 \} \). For SLACK: \( \{ p_j \} = \{ 7, 6, 5, 4, 4, 2 \} \).