The sandwich assembly-line problem

Part 1 due on November 25th; Part 2 due on December 9th

1 Introduction

The ubiquitous sandwich store is something almost all of us may be familiar with. There is a cash register where you order your sandwich. The operator gives a ticket to somebody which will start to assemble your sandwich. That employee adds the ingredients in front of him, and passes a partially assembled sandwich to the next worker, who will picked the sandwich up and add additional ingredients to the sandwich. Finally, the last employee will wrap the sandwich up and give it to you.

2 Model

In this project, we are interested in the generalized sandwich assembly problem. There are n tasks to execute using m processors. All the processors are identical, but tasks are not. Each task has a processing time p_i. The problem is to assign for each processor j an interval of tasks $I_j = [b_j; e_j]$. The intervals must be a partition of all the tasks: that is to say $\bigcup_{i=1}^{j} I_j = [1; n]$ and $\forall j \forall k, I_j \cap I_k = \emptyset$.

Since it is an assembly line, the throughput of the system (number of sandwiches produced per time unit) will be limited by the most loaded processor. The load of processor j is $L_j = \sum_{i \in I_j} p_i$. The most loaded processor has a load of $L = \max_j L_j$.

The point of the project is to design and implement algorithms to minimize L.

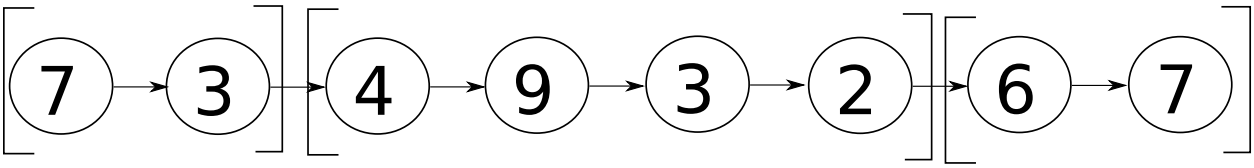

Figure 1: An instance of the problem and a valid solution with 3 processors. The most loaded processor is processor two and has a load of 18. (This solution is not optimal.)

3 Your task

You need to develop four algorithms to solve this problem; two for the Part 1 assignment, and two for the Part 2 assignment.

For **each** algorithm, you will need

1. the algorithm in pseudocode,
2. an implementation of the algorithm according to the specifications below,
3. and **report information/analysis** explaining the algorithms, including
   (a) proving that the optimal algorithms are optimal and
   (b) providing and proving their complexity as a function of the input of the problem.

The algorithms listed in the project are to be implemented in one of the three following programming languages: JAVA, C++ or C. Make sure to include how to compile and run the code with the Oracle JDK (for JAVA) or a standard GNU toolchain (C++ or C).

File format: The first line of the file indicates how many tasks there are. Each subsequent line give the processing time of a given task. An instance with 3 tasks of processing time 10, 11 and 12 will be:

```
3
10
11
12
```

Take the number of processors as a command line argument. Output the partition, as well as L.

Some sets of tasks for the problem can be found at `http://webpages.uncc.edu/~esaule/ITCS6114`. But feel free to create your own set of instance to study the problem better.

3.1 Assignment: Part 1

1. Brute force algorithm: design a brute-force algorithm to find the optimal solution.
2. Dynamic programming algorithm: design an dynamic programming algorithm to find the optimal solution.

3.2 Assignment: Part 2

1. Parametric Search
   (a) Design a greedy algorithm that decides whether there is a solution with a load $<= L^{target}$ and constructs such a solution if there is one. What is its complexity?
   (b) Notice that for all L^{target} values less than the optimal, the greedy algorithm will answer 'No' and that it will construct a solution for all the values greater than (or equal to) the optimal. Use this observation to find the optimal solution in $O(log(\sum p_i))$ calls to the greedy algorithm.
2. Design a heuristic algorithm to find a non-optimal, but reasonable solution, in a much shorter time. (Recall that greedy algorithms are typically the fastest. But any kind of algorithm is acceptable.)
3. **The final Part 2 report:** Study the experimental performance of all four algorithms on the different instances to answer multiple questions, including the following: How does the performance change when the number of processors vary? What is the impact of the number of tasks? Which algorithm is faster in practice? How good is the heuristic? Additionally, include the **report information/analysis** from each of the four algorithms in this report.

4 Extra Credit

4.1 Prefix Sum Array

The prefix sum array P of an array A of size n is an array of size $n+1$ such that $P[i] = \sum_{j<i} A[j]$. Explain how a prefix sum array can help speeding up the algorithms you developed for the sandwich assembly-line problem. How does the complexity of the algorithm change? Verify experimentally the difference of performance.

4.2 Heterogeneity

What if not all the processors are identical? Assume that processor i goes at speed s_i so that handling task j take $\frac{p_j}{s_i}$. If we enforce that processor 1 handles the first tasks of the chain, processor 2 takes the next ones, ..., can we adapt the previous algorithms to solve the problem? If we do not make this assumption (that is to say, any processor can take the first tasks), can we adapt the previous algorithms? Provide algorithms and justifications only, no implementation is needed.