Problem vs. Instance vs. Algorithm vs. Solution

- **Problem:** minimize makespan on 2 machines $P||C_{\text{max}}$
- **Instance:** 5 jobs with processing times (4, 1, 8, 5, 6)
- **Algorithm:** Alternate putting the jobs on machine 1 and machines 2.
- **Solution:** Machine 1 has jobs $J_1, J_3, J_5$ with total processing time 18, machine 2 has jobs $J_2, J_4$ with total processing time 6. Makespan is 18.

Goals:

- We want to develop algorithms that, on “any” instance, will produce good solutions.
- We want to understand how our algorithms perform, so that, given a new instance, we can predict how long they will take and what kind of solution they return.
Basics of Algorithm Analysis

Running Time: Given an algorithm, and an input of size $n$, we wish to know the running time as a function of $n$.

- We measure running time as a function of $n$, the size of the input (in bytes assuming a reasonable encoding).
- We work in the RAM model of computation. All “reasonable” operations take “1” unit of time. (e.g. +, *, -, /, array access, pointer following, writing a value, one byte of I/O...)

What is the running time of an algorithm

- Best case (seldom used)
- Average case (used if we understand the average)
- Worst case (used most often)

We measure as a function of $n$, and ignore low order terms.

- $5n^3 + n - 6$ becomes $n^3$
- $8n \log n - 60n$ becomes $n \log n$
- $2^n + 3n^4$ becomes $2^n$
Asymptotic notation

**big-O**

\[ O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \} \]

Alternatively, we say

\[ f(n) = O(g(n)) \text{ if there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \]

Informally, \( f(n) = O(g(n)) \) means that \( f(n) \) is asymptotically less than or equal to \( g(n) \).

**Classification** We use these to classify algorithms into classes, e.g. \( n, n^2, n \log n, 2^n \).
Simple Rules

- Nested loops multiply (even when the inner loop is from 1 to the outer loop value).
- Sequential loops add
- Repeated halving is linear.
Examples

FindMax \((A, n)\)

1 \hspace{.5em} \text{// } A \text{ is an array of length } n
2 \hspace{.5em} \text{maxval} = A[1]
3 \hspace{.5em} \text{for } i = 2 \text{ to } n
4 \hspace{3.5em} \text{if } (A[i] > \text{maxval})
5 \hspace{6em} \text{maxval} = A[i]
6 \hspace{.5em} \text{return } \text{maxval}

Running time is \(O(n)\)
Examples

**MatMult** \((A, B, C, p, q, r)\)

1 // **A** is \(p \times q\); **B** is \(q \times r\); **C** is \(p \times r\)
2 for \(i = 1\) to \(p\)
3 \hspace{1em} for \(j = 1\) to \(r\)
4 \hspace{2em} \(C[i, j] = 0\)
5 \hspace{1em} for \(k = 1\) to \(q\)
6 \hspace{2em} \(C[i, j] = C[i, j] + A[i, k] \times B[k, j]\)

Running Time is \(O(pqr)\). If matrices are \(n \times n\), then \(O(n^3)\).
Some Things to Know

- Sorting $n$ numbers takes $O(n \log n)$ time.
- Finding a shortest path with non-negative weights in a graph with $n$ nodes and $m$ edges takes $O(m \log n)$ time.
- You can use a Priority Queue Data Structure to maintain a set of $n$ numbers, and insert and delete the maximum in $O(\log n)$ time per operation.