Kapitel 4:
Approximation Algorithms

Approximation Algorithms

Q. Suppose I need to solve an NP-hard problem. What should I do?
A. Theory says you're unlikely to find a poly-time algorithm.

Must sacrifice one of three desired features.

- Solve problem to optimality.
- Solve problem in poly-time.
- Solve arbitrary instances of the problem.

ρ-approximation algorithm.

- Guaranteed to run in poly-time.
- Guaranteed to solve arbitrary instance of the problem
- Guaranteed to find solution within ratio ρ of true optimum.

Challenge. Need to prove a solution's value is close to optimum, without even knowing what optimum value is!

Kapitel 4: Approximation Algorithms

Inhalt:

- Greedy Techniques
  - Load-Balancing Problem
  - Center Selection Problem
- Pricing Method
  - Vertex Cover Problem
- Linear Programming and Rounding
  - Vertex Cover Problem
  - Generalized Load-Balancing Problem
- Polynomial Time Approximation Scheme
  - Knapsack Problem

Load Balancing

Input. m identical machines; n jobs, job j has processing time t_j.

- Job j must run contiguously on one machine.
- A machine can process at most one job at a time.

Def. Let J(i) be the subset of jobs assigned to machine i. The load of machine i is $L_i = \Sigma_{j \in J(i)} t_j$.

Def. The makespan is the maximum load on any machine $L = \max_i L_i$.

Load balancing problem. Assign each job to a machine to minimize makespan.

Load Balancing: List Scheduling

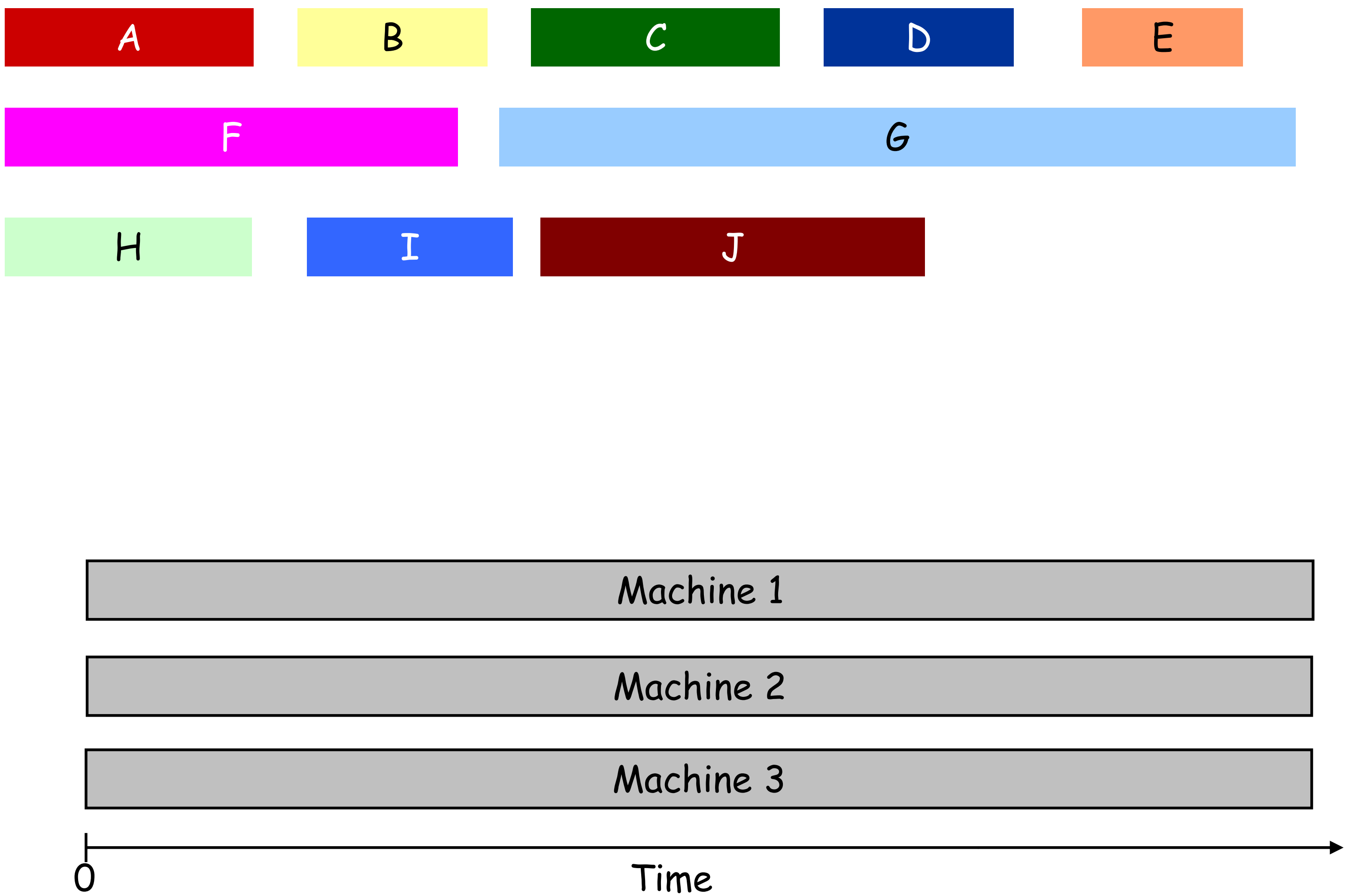
List-scheduling algorithm.

- Consider n jobs in some fixed order.
- Assign job j to machine whose load is smallest so far.

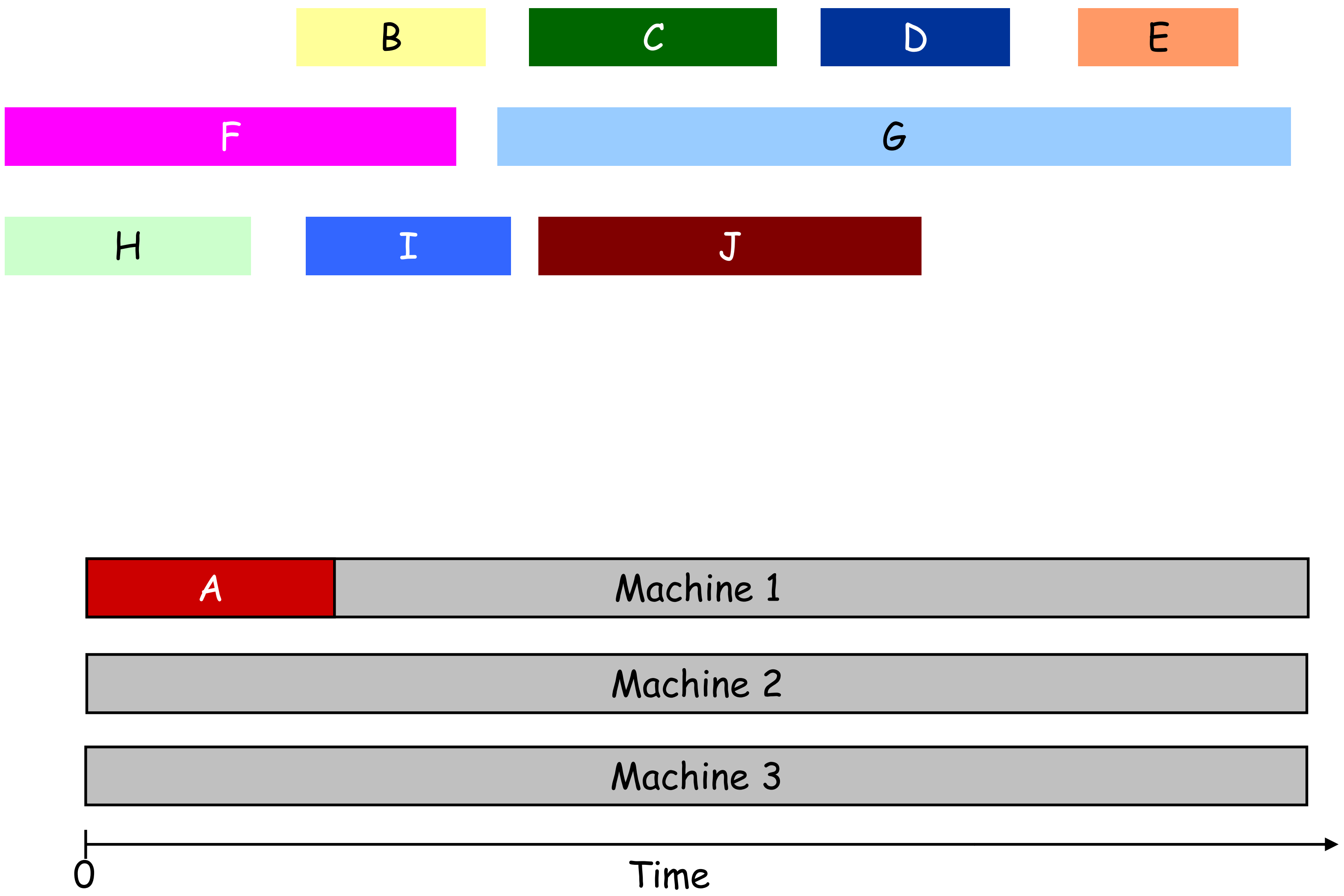
```
List-Scheduling(m, n, t_1,t_2,...,t_n) {
   for i = 1 to m {
      L_i ← 0            ← load on machine i
      J(i) ← φ           ← jobs assigned to machine i
   }

   for j = 1 to n {
      i = argmin_k L_k              ← machine i has smallest load
      J(i) ← J(i) ∪ {j}             ← assign job j to machine i
      L_i ← L_i + t_j               ← update load of machine i
   }
}
```

Implementation. $O(n \log m)$ using a priority queue.

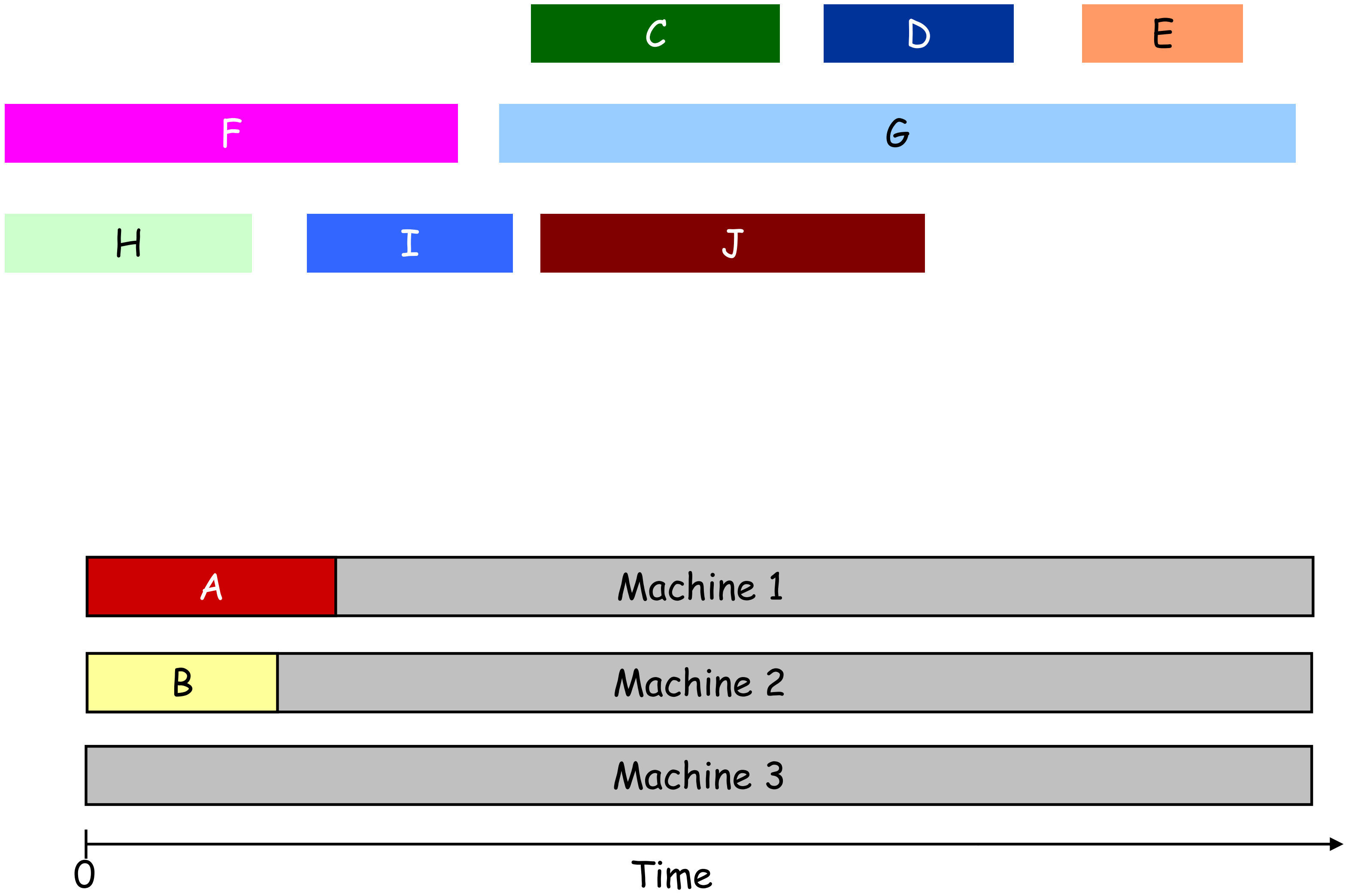
Load Balancing: List Scheduling



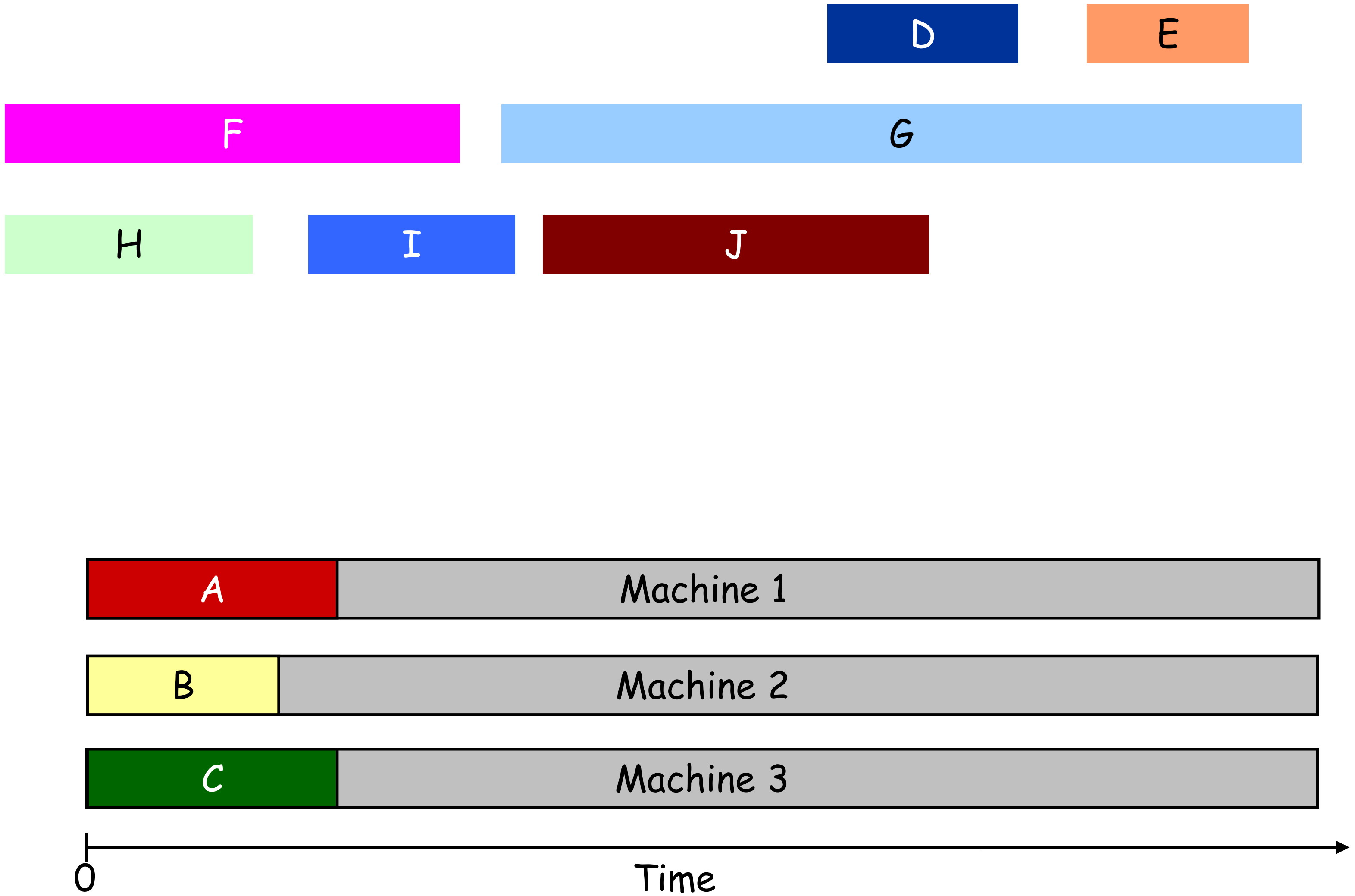
Load Balancing: List Scheduling



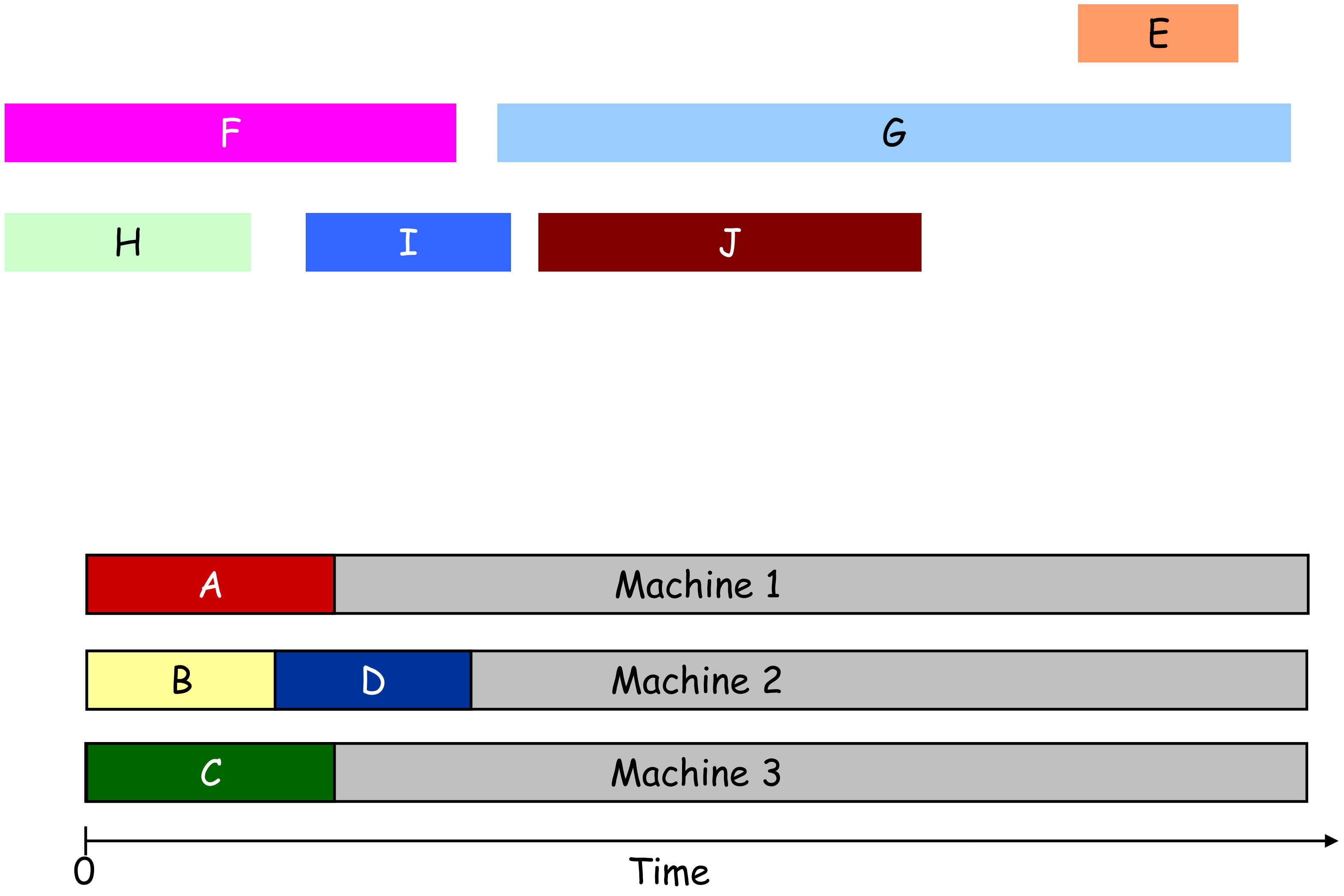
Load Balancing: List Scheduling



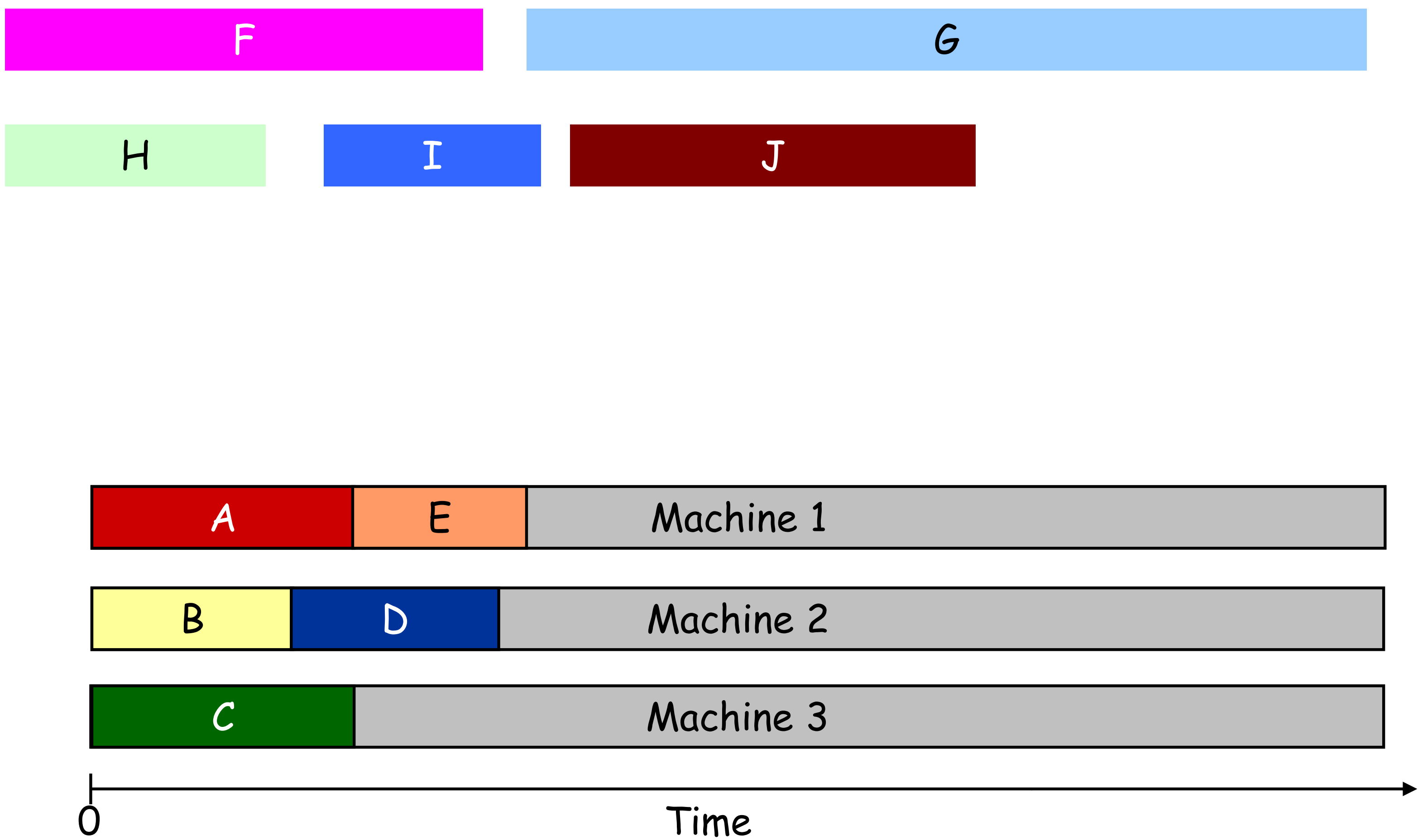
Load Balancing: List Scheduling



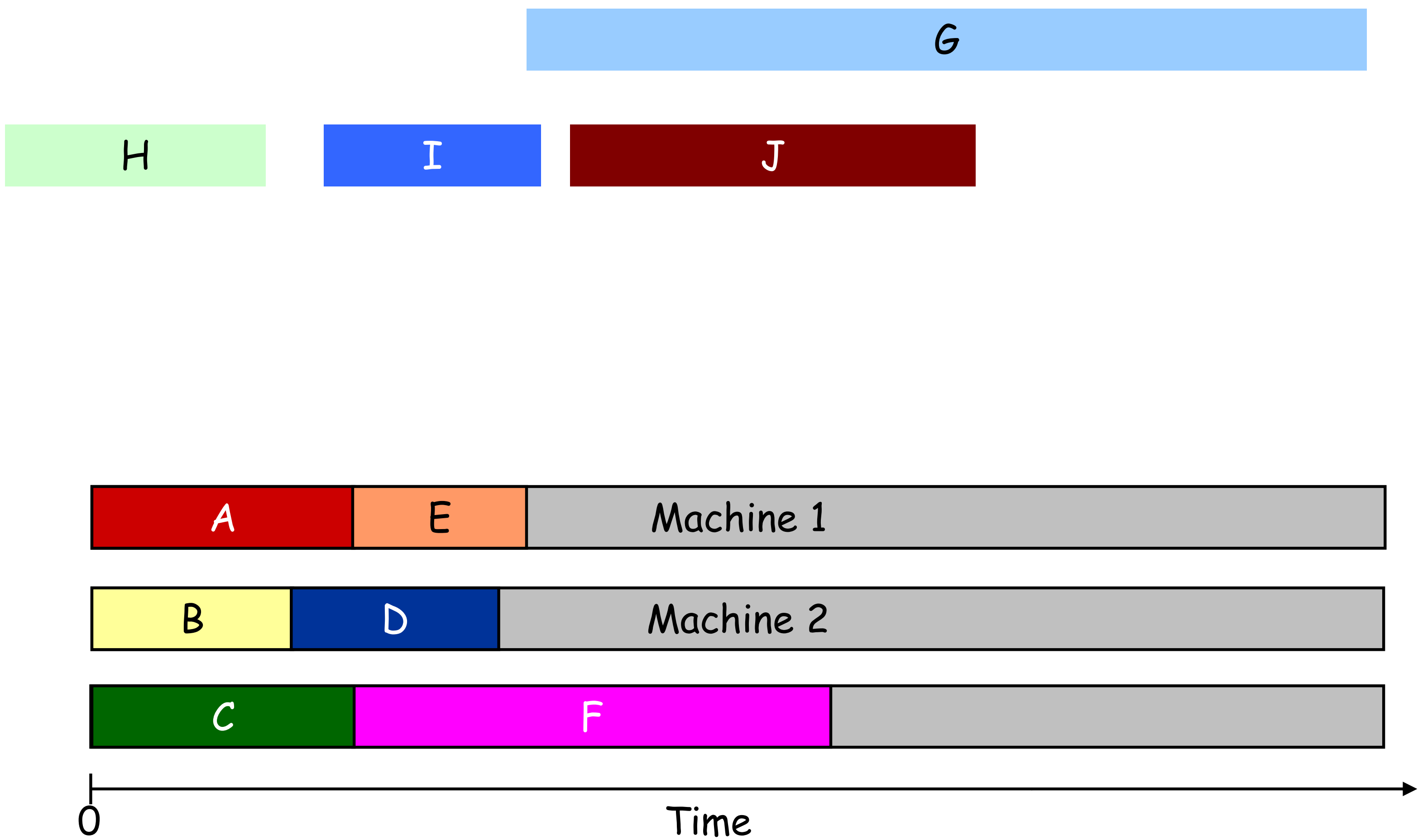
Load Balancing: List Scheduling



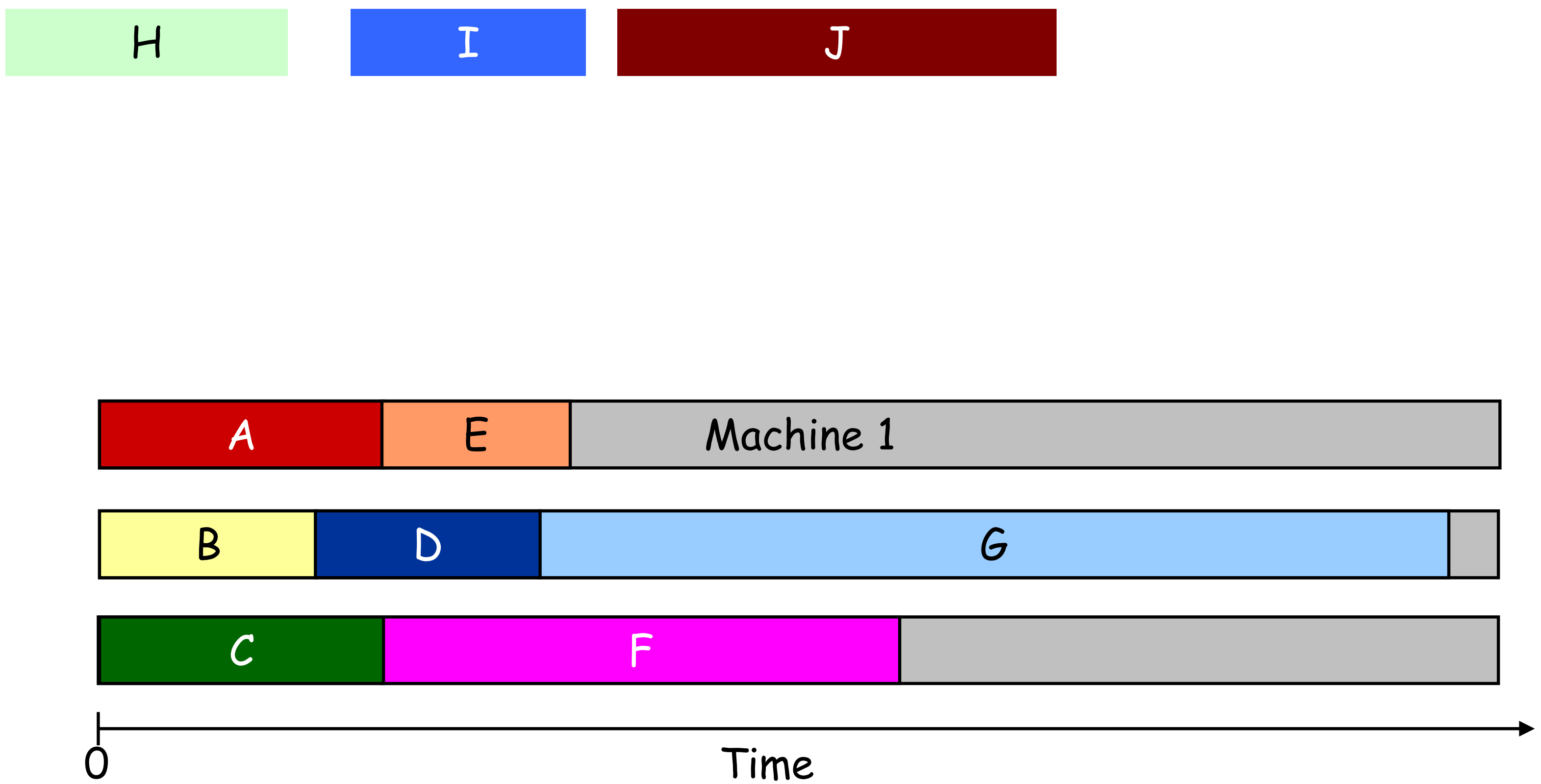
Load Balancing: List Scheduling



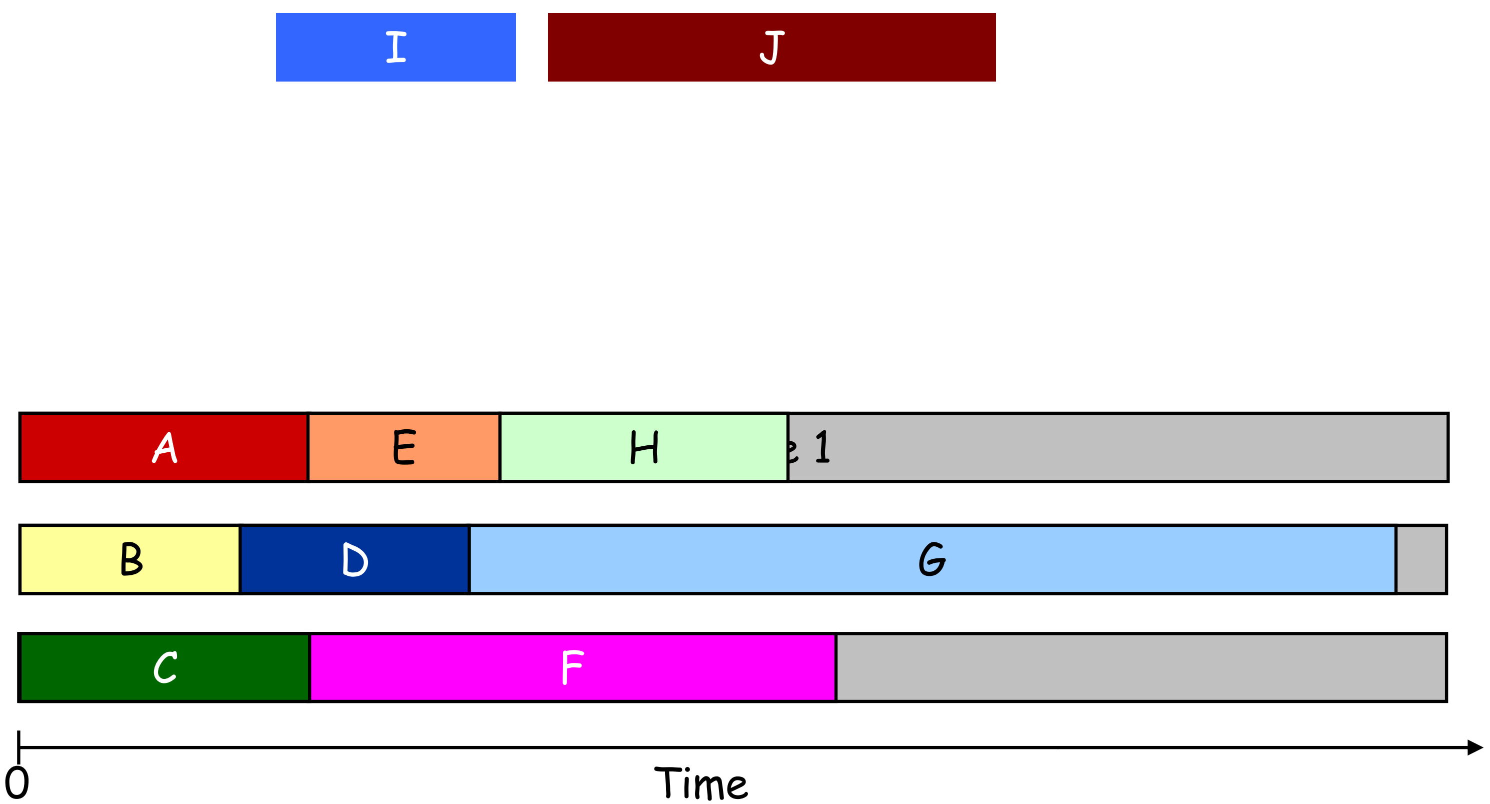
Load Balancing: List Scheduling



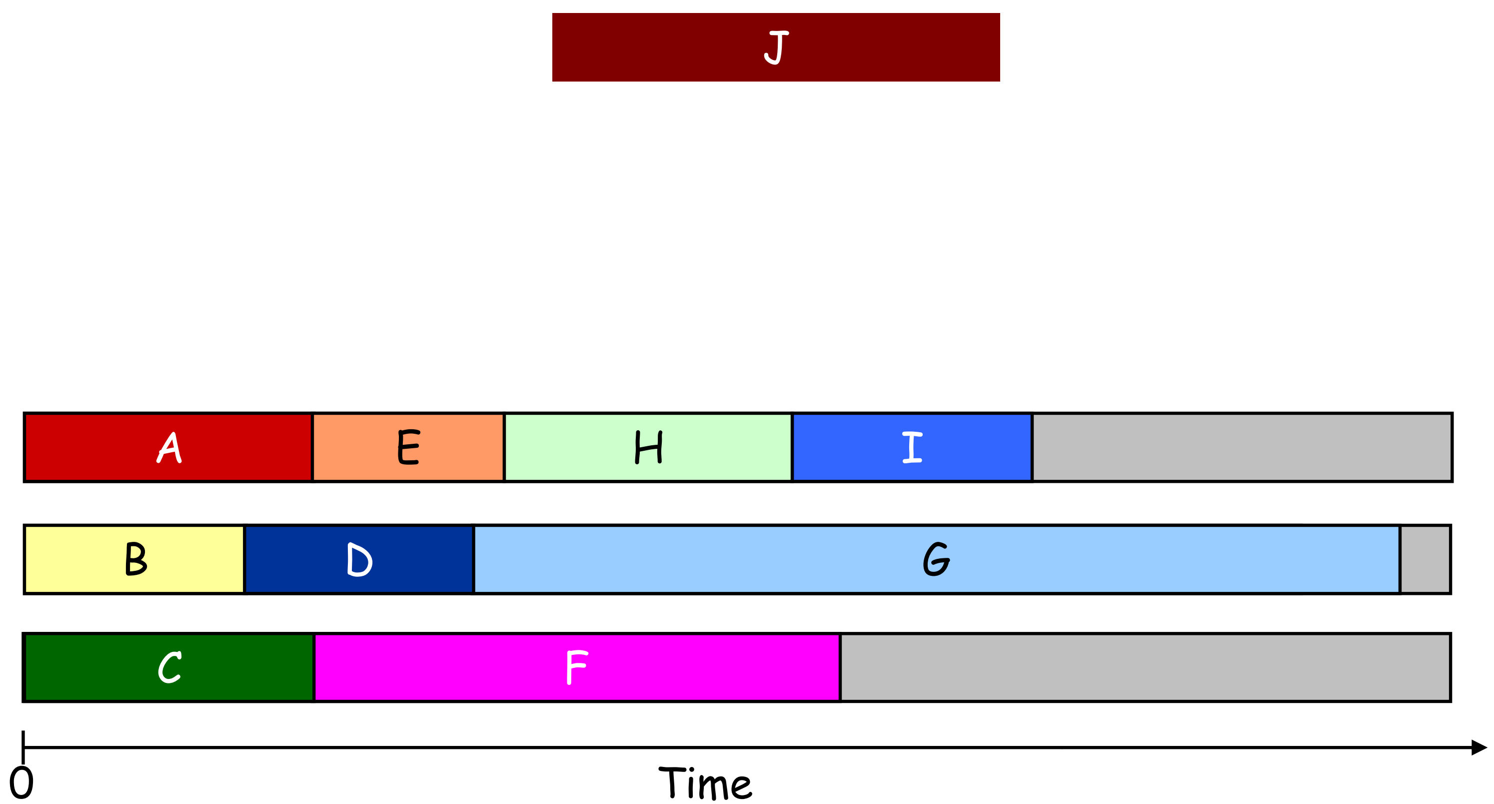
Load Balancing: List Scheduling



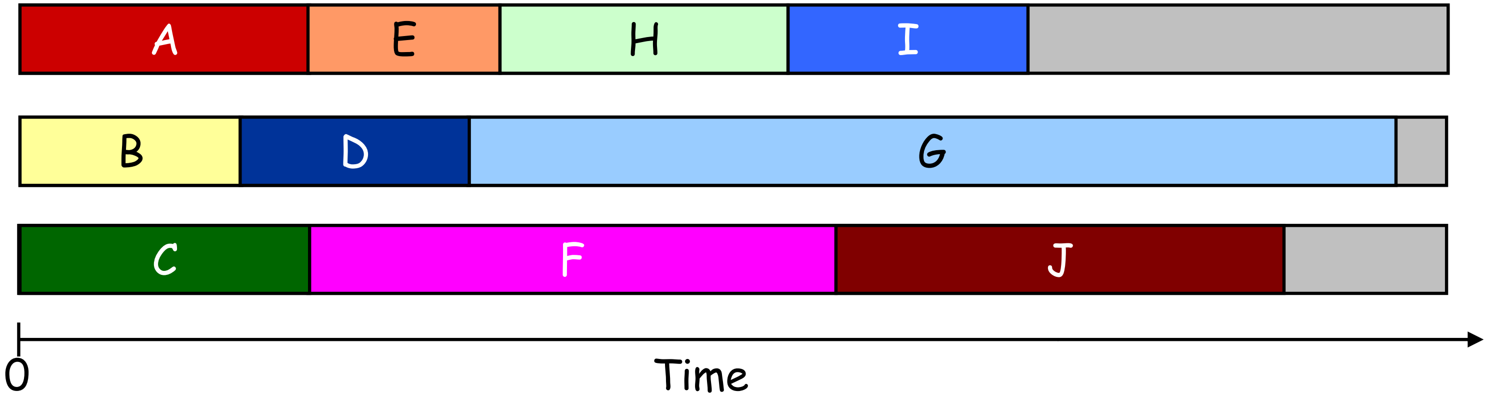
Load Balancing: List Scheduling



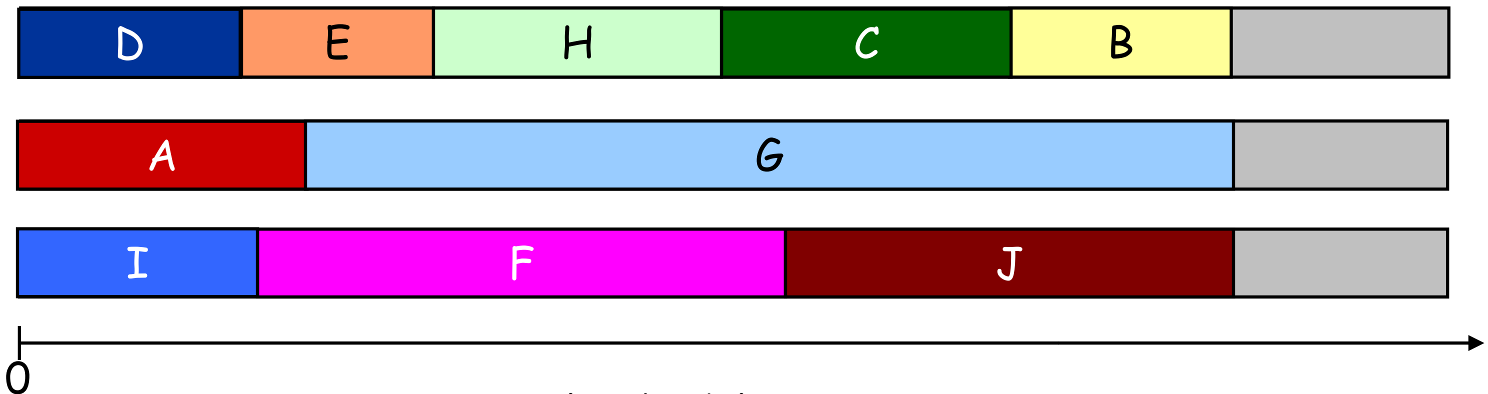
Load Balancing: List Scheduling



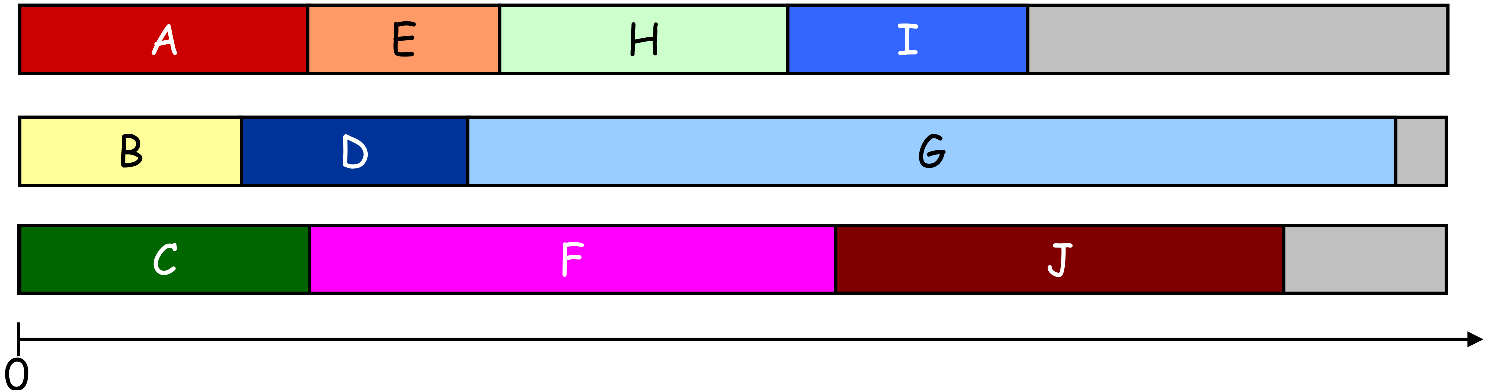
Load Balancing: List Scheduling



Load Balancing: List Scheduling



Optimal Schedule



List schedule

Load Balancing: List Scheduling Analysis

Theorem. [Graham, 1966] Greedy algorithm is a 2-approximation.

- First worst-case analysis of an approximation algorithm.
- Need to compare resulting solution with optimal makespan L*.

Lemma 1. The optimal makespan $L^* \geq \max_j t_j$.
Pf. Some machine must process the most time-consuming job. ▪

Lemma 2. The optimal makespan $L^* \geq \frac{1}{m} \sum_j t_j$.
Pf.

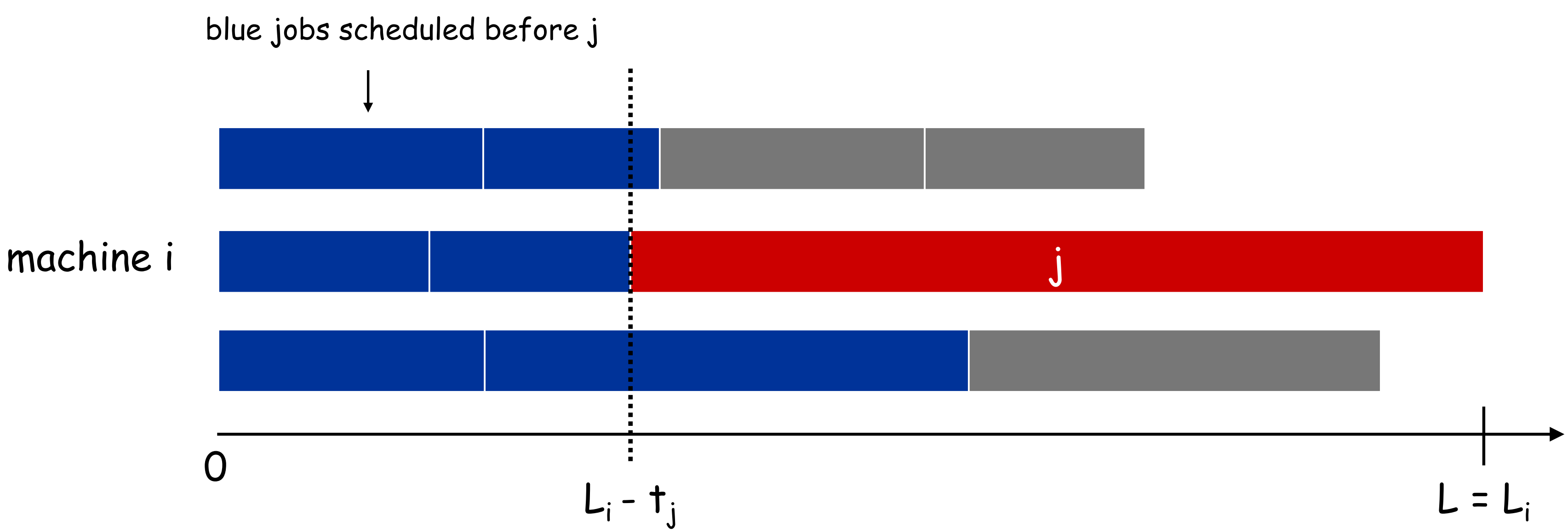
- The total processing time is $\Sigma_j t_j$.
- One of m machines must do at least a 1/m fraction of total work. ▪

Load Balancing: List Scheduling Analysis

Theorem. Greedy algorithm is a 2-approximation.

Pf. Consider load L_i of bottleneck machine i.

- Let j be last job scheduled on machine i.
- When job j assigned to machine i, i had smallest load. Its load before assignment is $L_i - t_j \Rightarrow L_i - t_j \leq L_k$ for all $1 \leq k \leq m$.



Load Balancing: List Scheduling Analysis

Theorem. Greedy algorithm is a 2-approximation.

Pf. Consider load L_i of bottleneck machine i.

- Let j be last job scheduled on machine i.
- When job j assigned to machine i, i had smallest load. Its load before assignment is $L_i - t_j \Rightarrow L_i - t_j \leq L_k$ for all $1 \leq k \leq m$.
- Sum inequalities over all k and divide by m:

$$\begin{aligned} L_i - t_j &\leq \tfrac{1}{m} \textstyle\sum_k L_k \\ &= \tfrac{1}{m} \textstyle\sum_k t_k \\ \text{Lemma 2} \longrightarrow \quad &\leq L^* \end{aligned}$$

- Now $$L_i = \underbrace{(L_i - t_j)}_{\leq L^*} + \underbrace{t_j}_{\leq L^*} \leq 2L^*. \quad \blacksquare$$

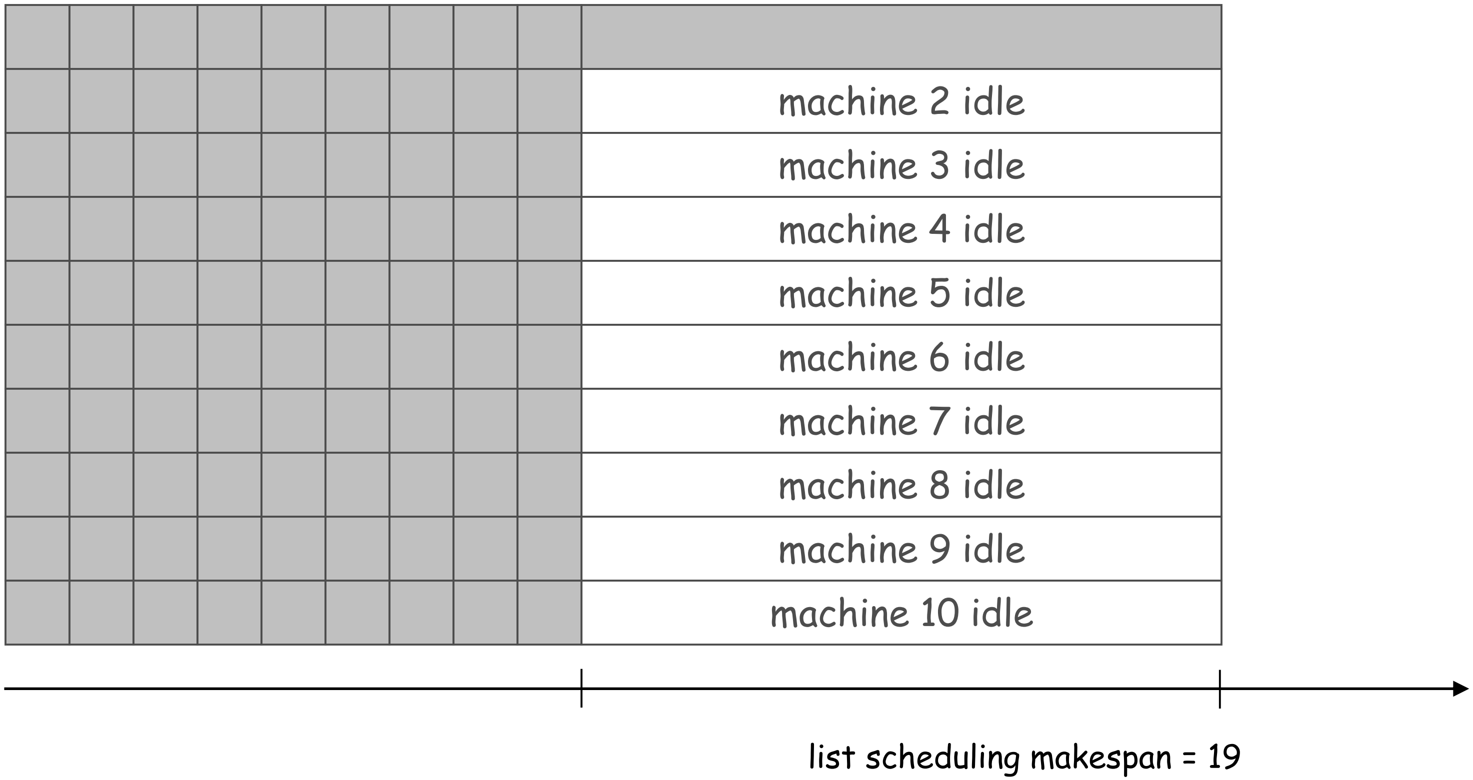
Lemma 1

Load Balancing: List Scheduling Analysis

Q. Is our analysis tight?
A. Essentially yes.

Ex: m machines, m(m-1) jobs of length 1, one job of length m

m = 10

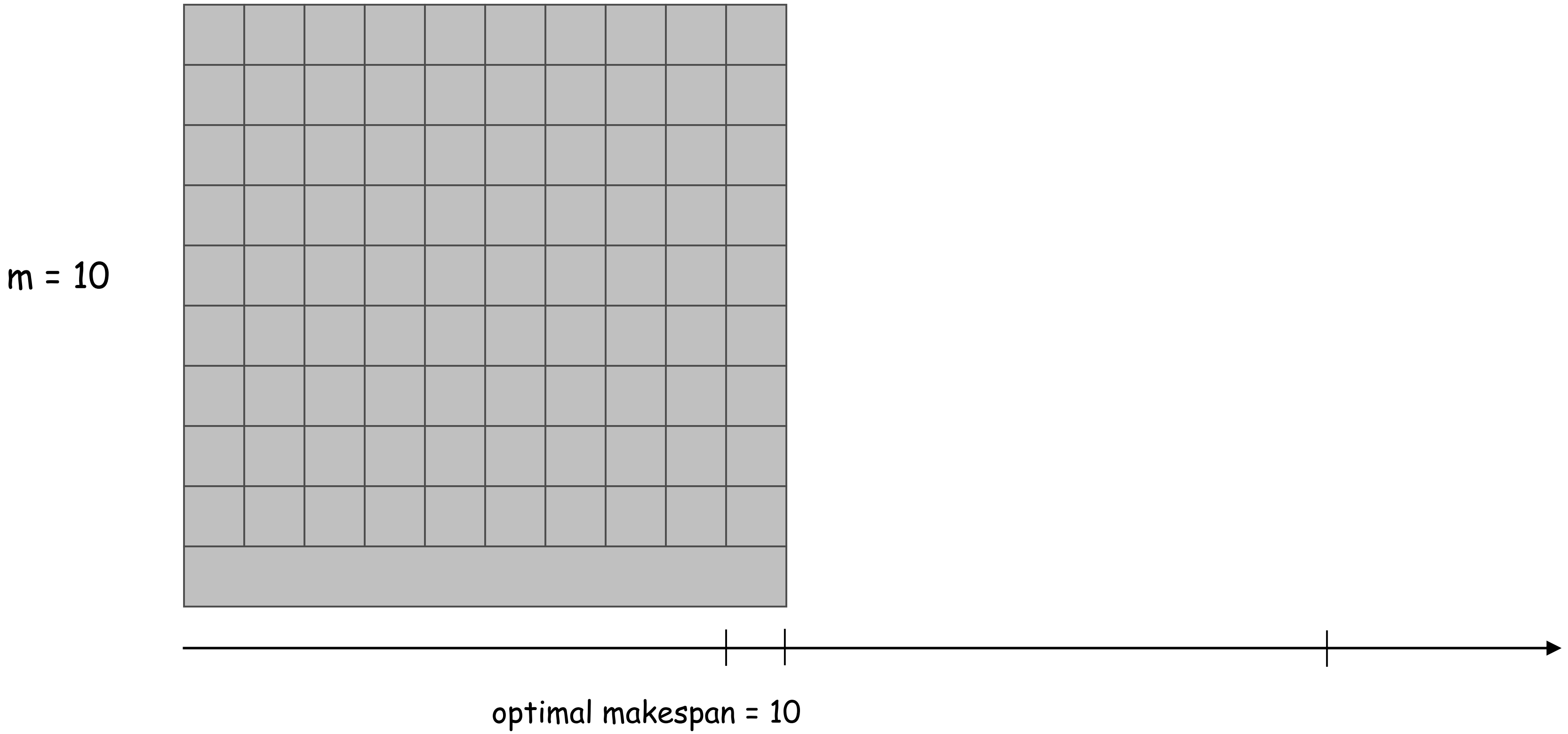


Load Balancing: List Scheduling Analysis

Q. Is our analysis tight?

A. Essentially yes.

Ex: m machines, m(m-1) jobs of length 1, one job of length m



Load Balancing: LPT Rule

Longest processing time (LPT). Sort n jobs in descending order of processing time, and then run list scheduling algorithm.

```
LPT-List-Scheduling(m, n, t1,t2,...,tn) {
   Sort jobs so that t1 ≥ t2 ≥  ... ≥ tn

   for i = 1 to m {
      Li ← 0            ← load on machine i
      J(i) ← φ          ← jobs assigned to machine i
   }

   for j = 1 to n {
      i = argmink Lk          ← machine i has smallest load
      J(i) ← J(i) ∪ {j}       ← assign job j to machine i
      Li ← Li + tj            ← update load of machine i
   }
}
```

Load Balancing: LPT Rule

Observation. If at most m jobs, then list-scheduling is optimal.
Pf. Each job put on its own machine. ▪

Lemma 3. If there are more than m jobs, $L^* \geq 2\, t_{m+1}$.
Pf.
- Consider first m+1 jobs $t_1, \ldots, t_{m+1}$.
- Since the t_i's are in descending order, each takes at least t_{m+1} time.
- There are m+1 jobs and m machines, so by pigeonhole principle, at least one machine gets two jobs. ▪

Theorem. LPT rule is a 3/2 approximation algorithm.
Pf. Same basic approach as for list scheduling.

$$L_i = \underbrace{(L_i - t_j)}_{\leq L^*} + \underbrace{t_j}_{\leq \frac{1}{2} L^*} \leq \tfrac{3}{2} L^*. \quad ▪$$

Lemma 3
(by observation, can assume number of jobs > m)

Load Balancing: LPT Rule

Q. Is our 3/2 analysis tight?
A. No.

Theorem. [Graham, 1969] LPT rule is a 4/3-approximation.
Pf. More sophisticated analysis of same algorithm.

Q. Is Graham's 4/3 analysis tight?
A. Essentially yes.

Ex: m machines, $n = 2m+1$ jobs, 2 jobs of length $m+1$, $m+2$, …, $2m-1$, $2m$ and one job of length m.

Kapitel 4: Approximation Algorithms

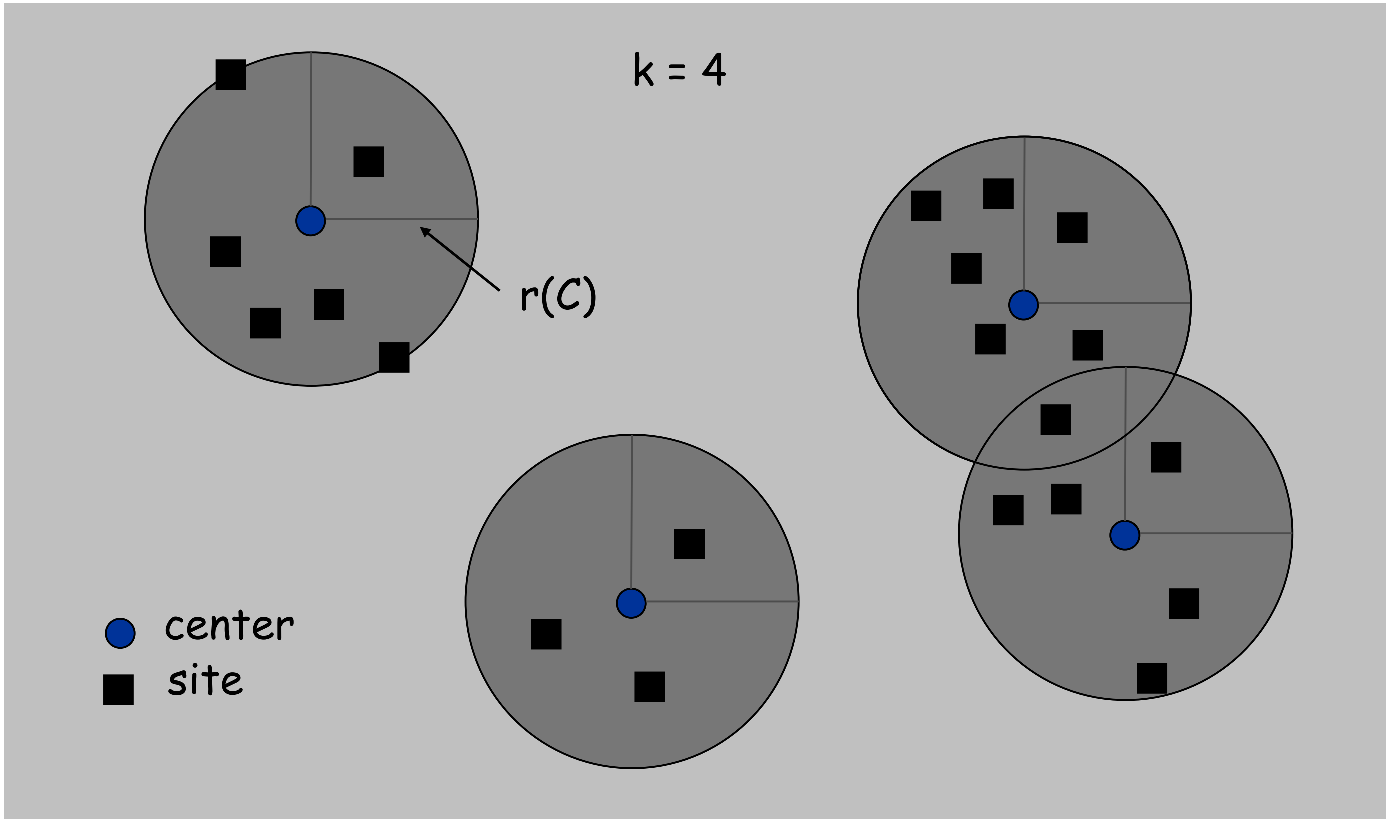
Inhalt:

- Greedy Techniques
  - Load-Balancing Problem
  - Center Selection Problem
- Pricing Method
  - Vertex Cover Problem
- Linear Programming and Rounding
  - Vertex Cover Problem
  - Generalized Load-Balancing Problem
- Polynomial Time Approximation Scheme
  - Knapsack Problem

Center Selection Problem

Input. Set of n sites $s_1, \ldots, s_n$ and integer $k > 0$.

Center selection problem. Select k centers C so that maximum distance from a site to nearest center is minimized.



Center Selection Problem

Input. Set of n sites $s_1, \ldots, s_n$ and integer $k > 0$.

Center selection problem. Select k centers C so that maximum distance from a site to nearest center is minimized.

Notation.

- $dist(x, y)$ = distance between x and y.
- $dist(s_i, C) = \min_{c \in C} dist(s_i, c)$ = distance from s_i to closest center.
- $r(C) = \max_i dist(s_i, C)$ = smallest covering radius.

Goal. Find set of centers C that minimizes $r(C)$, subject to $|C| = k$.

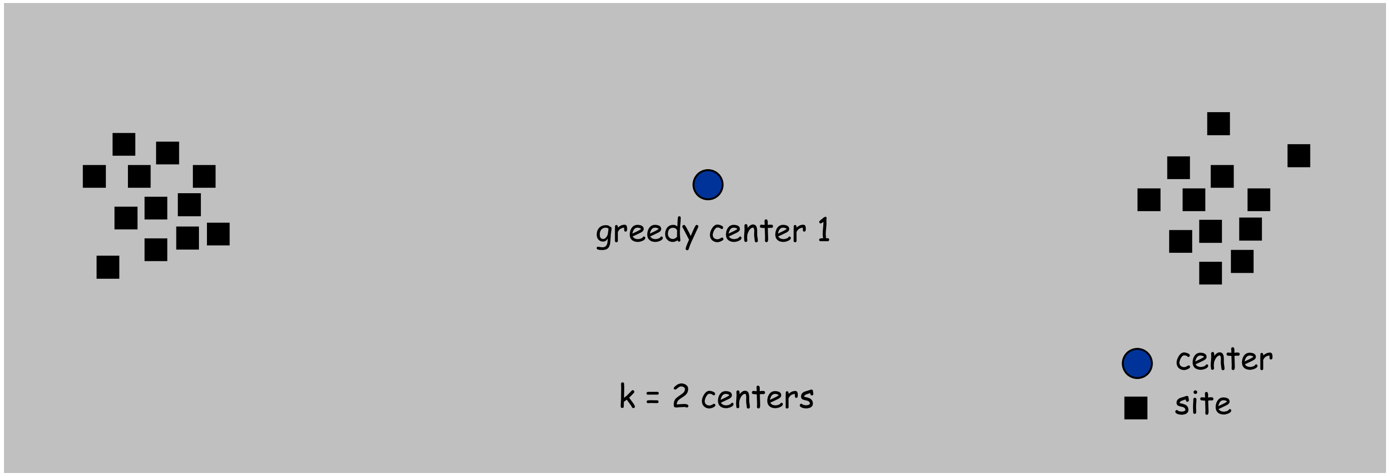
Distance function properties.

- $dist(x, x) = 0$ (identity)
- $dist(x, y) = dist(y, x)$ (symmetry)
- $dist(x, y) \leq dist(x, z) + dist(z, y)$ (triangle inequality)

Greedy Algorithm: A False Start

Greedy algorithm. Put the first center at the best possible location for a single center, and then keep adding centers so as to reduce the covering radius each time by as much as possible.

Remark: arbitrarily bad!



Center Selection: Greedy Algorithm

Greedy algorithm. Repeatedly choose the next center to be the site farthest from any existing center.

```
Greedy-Center-Selection(k, n, s1,s2,…,sn) {

   C = φ
   repeat k times {
      Select a site si with maximum dist(si, C)
      Add si to C
   }
   return C
}
```

site farthest from any center

Observation. Upon termination all centers in C are pairwise at least $r(C)$ apart.
Pf. By construction of algorithm.

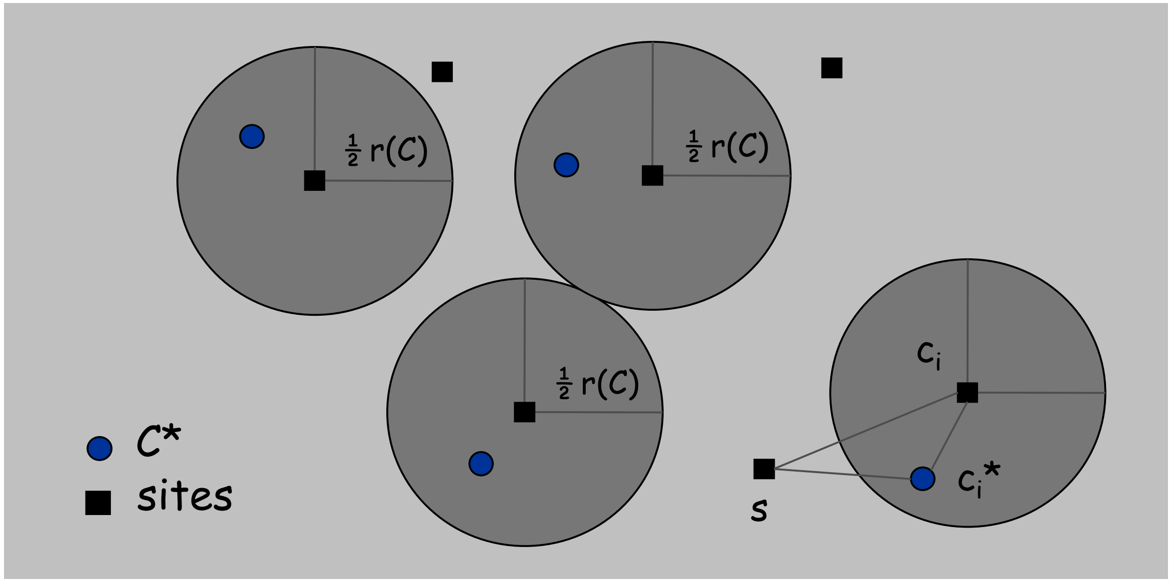
Center Selection: Analysis of Greedy Algorithm

Theorem. Let C^* be an optimal set of centers. Then $r(C) \leq 2r(C^*)$.

Pf. (by contradiction) Assume $r(C^*) < \frac{1}{2}\, r(C)$.

- For each site c_i in C, consider ball of radius $\frac{1}{2}\, r(C)$ around it.
- Exactly one c_i^* in each ball; let c_i be the site paired with c_i^*.
- Consider any site s and its closest center c_i^* in C^*.
- $\text{dist}(s, C) \leq \underbrace{\text{dist}(s, c_i)}_{\Delta\text{-inequality}} \leq \text{dist}(s, c_i^*) + \text{dist}(c_i^*, c_i) \leq 2r(C^*)$.
- Thus $r(C) \leq 2r(C^*)$. ▪

$\leq r(C^*)$ since c_i^* is closest center



Center Selection

Theorem. Let C^* be an optimal set of centers. Then $r(C) \leq 2r(C^*)$.

Theorem. Greedy algorithm is a 2-approximation for center selection problem.

Remark. Greedy algorithm always places centers at sites, but is still within a factor of 2 of best solution that is allowed to place centers anywhere.

e.g., points in the plane

Question. Is there hope of a 3/2-approximation? 4/3?

Theorem. Unless P = NP, there is no ρ-approximation for center-selection problem for any $\rho < 2$ and $k > 2$.

Center Selection: Hardness of Approximation

Theorem. Unless P = NP, there is no ρ-approximation algorithm for metric k-center problem for any $\rho < 2$.

Pf. We show how we could use a $(2 - \varepsilon)$ approximation algorithm for k-center to solve DOMINATING-SET in poly-time.

- Let $G = (V, E)$, k be an instance of DOMINATING-SET.
- Construct instance G' of k-center with sites V and distances
  - $d(u, v) = 1$ if $(u, v) \in E$
  - $d(u, v) = 2$ if $(u, v) \notin E$
- Note that G' satisfies the triangle inequality.
- Claim: G has dominating set of size k iff there exists k centers C^* with $r(C^*) = 1$.
- Thus, if G has a dominating set of size k, a $(2 - \varepsilon)$-approximation algorithm on G' must find a solution C^* with $r(C^*) = 1$ since it cannot use any edge of distance 2.

Fragen?