# 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.

$\rho$-approximation algorithm.

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

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

# Load Balancing

# 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 $L_i = \sum_{j \in J(i)} t_j$.

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

Load balancing. 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, 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 = argmin_k 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
    }
}
```

Implementation. O(n log n) using a priority queue.

# 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.

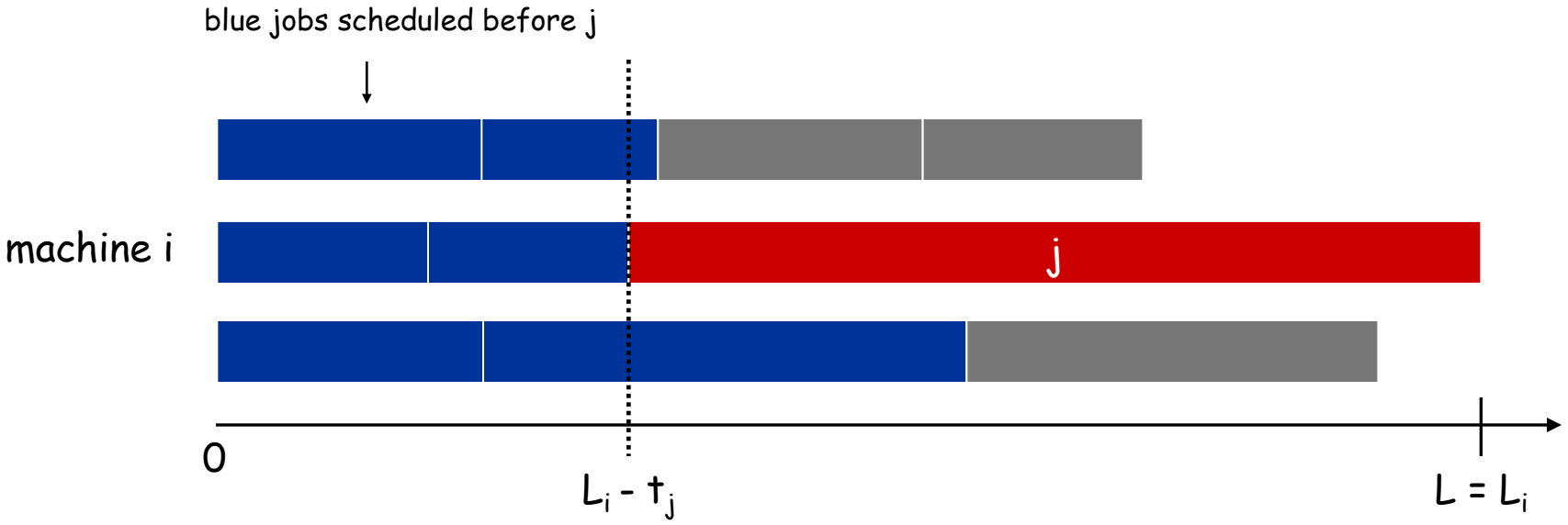
- The total processing time is $\Sigma_j t_j$ .
- One of m machines must do at least a 1/m fraction of total work. ▪

# 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:

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

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

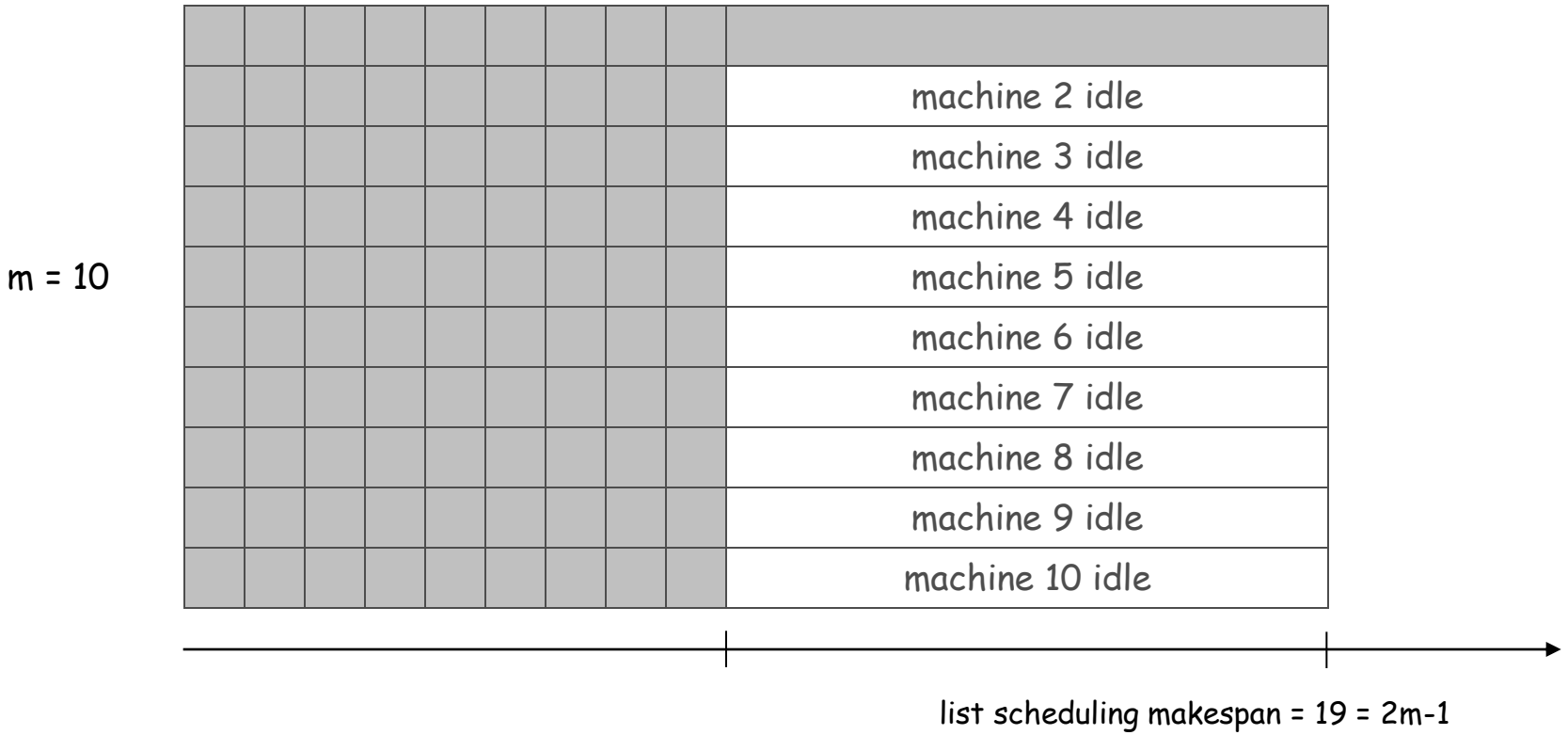
Lemma 2

# Load Balancing: List Scheduling Analysis

Q. Is our analysis tight?

A. Essentially yes.

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

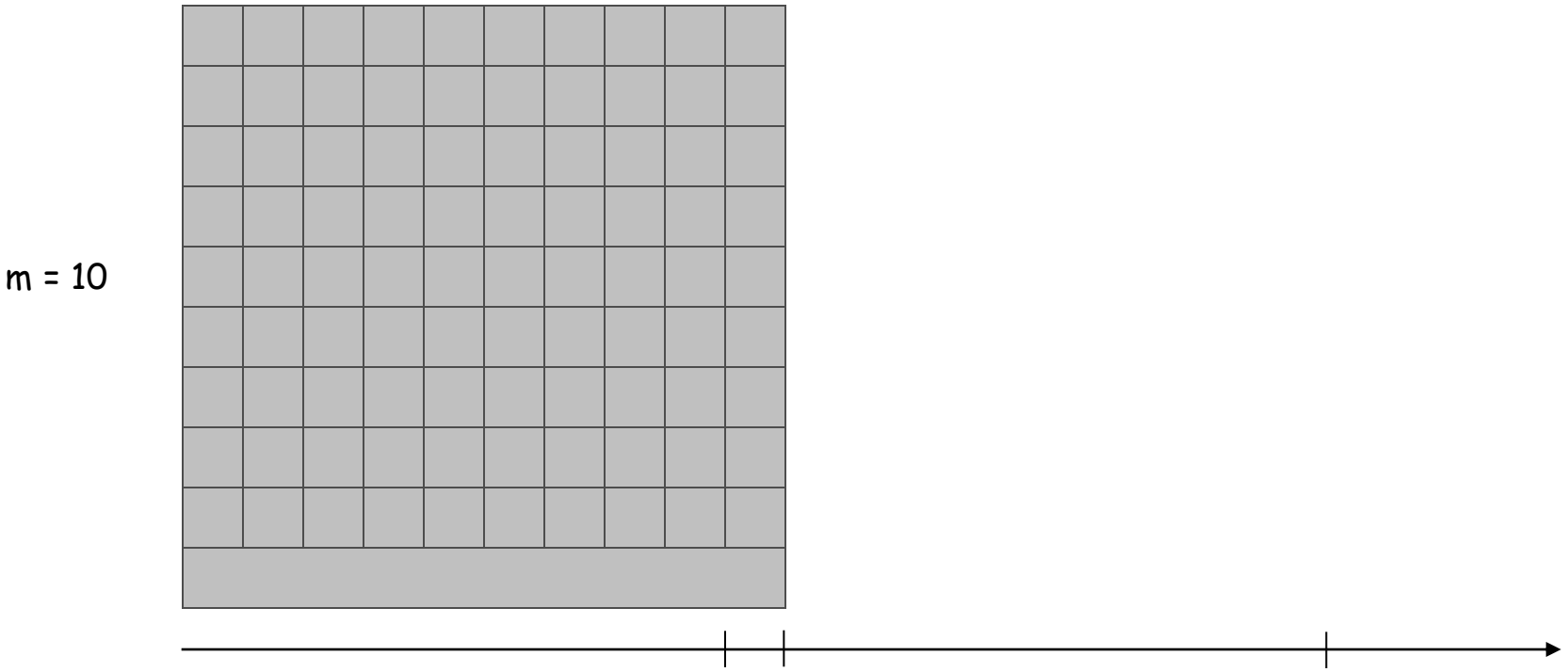


# Load Balancing: List Scheduling Analysis

Q. Is our analysis tight?

A. Essentially yes.

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




optimal makespan = 10 = m. Thus, ratio=(2m-1)/m ≈ 2 for large 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, t_1,t_2,...,t_n) {
    Sort jobs so that 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
    }
}
```

# 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.

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

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 and one job of length m.

# Set Cover: A general greedy heuristic

# Set Cover

Set Cover Problem is based on a set U of n elements and a list $S_1, \ldots,$ $S_m$ of subsets of U; we say that a set cover is a collection of these sets whose union is equal to all of U

In the version of the problem we consider here, each set $S_i$ has an associated weight $w_i \geq 0$.
The goal is to find a set cover C so that the total weight

$$\sum_{S_i \in \mathcal{C}} w_i$$

is minimized.

Note that this problem is at least as hard as the decision version of Set Cover we encountered earlier

# Set Cover greedy

The algorithm will have the property that it builds the cover one set at a time; to choose its next set, it looks for one that seems to make the most progress toward the goal.

What is a natural way to define "progress" in this setting?

Desirable sets have two properties: They have small weight $w_i$, and they cover lots of elements.

Neither of these properties alone, however, would be enough for designing a good approximation algorithm.

Instead, it is natural to combine these two criteria into the single measure $w_i / |S_i|$—that is, by selecting $S_i$, we cover $|S_i|$ elements at a cost of $w_i$, and so this ratio gives the "cost per element covered," a very reasonable thing to use as a guide.

# Set Cover greedy

Of course, once some sets have already been selected, we are only concerned with how we are doing on the elements still left uncovered.

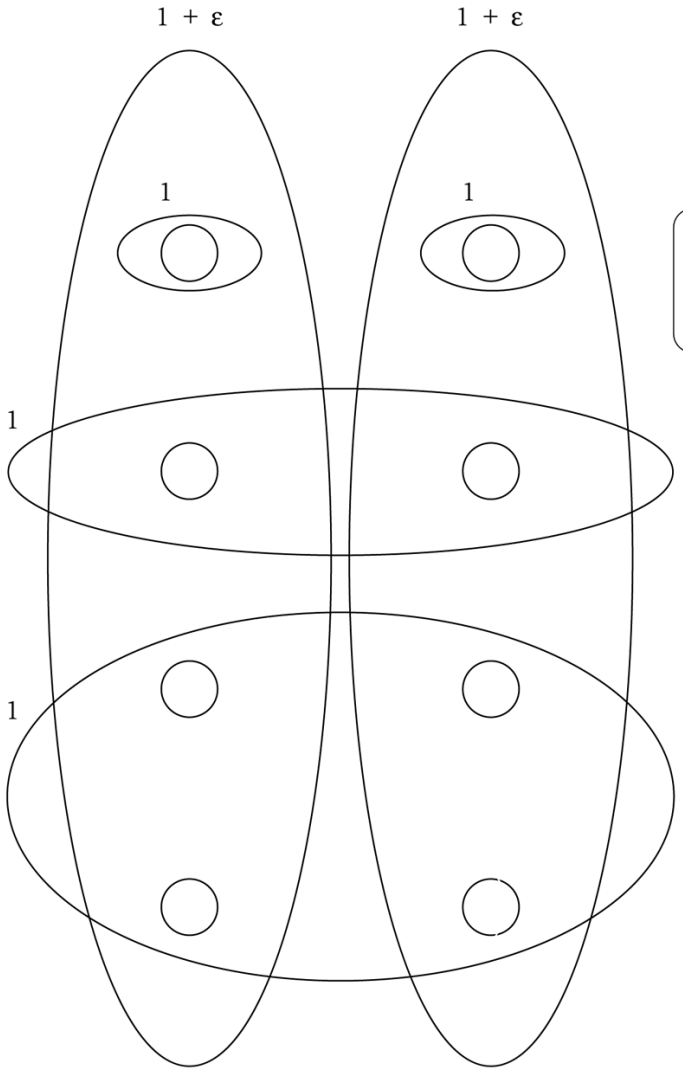
So we will maintain the set $R$ of remaining uncovered elements and choose the set $S_i$ that minimizes $w_i/|S_i \cap R|$.

---

Greedy-Set-Cover:
Start with $R = U$ and no sets selected
While $R \neq \emptyset$
  Select set $S_i$ that minimizes $w_i/|S_i \cap R|$
  Delete set $S_i$ from $R$
EndWhile
Return the selected sets

---

# Set Cover greedy algorithm (bad) example




Two sets can be used to cover everything, but the greedy algorithm doesn't find them.

**Figure 11.6** An instance of the Set Cover Problem where the weights of sets are either 1 or $1+\varepsilon$ for some small $\varepsilon > 0$. The greedy algorithm chooses sets of total weight 4, rather than the optimal solution of weight $2+2\varepsilon$.

# Set Cover greedy algorithm analysis

Recall the intuitive meaning of the ratio $w_i / |S_i \cap R|$ used by the algorithm; it is the "cost paid" for covering each new element. Let's record this cost paid for element $s$ in the quantity $c_s$.

We add the following line to the code immediately after selecting the set $S_i$.

---

Define $c_s = w_i/|S_i \cap R|$ for all $s \in S_i \cap R$

---

The values $c_s$ do not affect the behavior of the algorithm at all; we view them as a bookkeeping device to help in our comparison with the optimum $w^*$.

As each set $S_i$ is selected, its weight is distributed over the costs $c_s$ of the elements that are newly covered.

Thus these costs completely account for the total weight of the set cover, and so we have

**(11.9)** *If $\mathcal{C}$ is the set cover obtained by* `Greedy-Set-Cover`, *then* $\sum_{S_i \in \mathcal{C}} w_i = \sum_{s \in U} c_s$.

# Set Cover greedy algorithm analysis

The key to the analysis is to ask how much total cost any single set $S_k$ can account for—in other words, to give a bound on

$$\sum_{s \in S_k} c_s$$

relative to the weight $w_k$ of the set.
Giving an upper bound on the ratio

$$\frac{\sum_{s \in S_k} c_s}{w_k}$$

that holds for every set says, in effect, "To cover a lot of cost, you must use a lot of weight."We know that the optimum solution must cover the full cost $\sum_{s \in U} c_s$

via the sets it selects; so this type of bound will establish that it needs to use at least a certain amount of weight. This is a lower bound on the optimum, just as we need for the analysis.

# Set Cover greedy algorithm analysis

Our analysis will use the *harmonic function*

$$H(n) = \sum_{i=1}^{n} \frac{1}{i}.$$

*It is known that* $H(n) = \Theta(\ln n)$.

Here is the key to establishing a bound on the performance of the algorithm.

**(11.10)** *For every set* $S_k$, *the sum* $\sum_{s \in S_k} c_s$ *is at most* $H(|S_k|) \cdot w_k$.

Pf. To simplify the notation, we will assume that the elements of $S_k$ are the first $d = |S_k|$ elements of the set $U$; that is, $S_k = \{s_1, \ldots, s_d\}$.

Furthermore, let us assume that these elements are labeled in the order in which they are assigned a cost $c_{s_j}$ by the greedy algorithm (with ties broken arbitrarily).

There is no loss of generality in doing this, since it simply involves a renaming of the elements in $U$.

# Set Cover greedy algorithm analysis

Now consider the iteration in which element $s_j$ is covered by the greedy algorithm, for some $j \leq d$. At the start of this iteration, $s_j, s_{j+1}, \ldots, s_d \in R$ by our labeling of the elements.

This implies that $|S_k \cap R|$ is at least $d - j + 1$, and so the average cost of the set $S_k$ is at most

$$\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}$$

Note that this is not necessarily an equality, since $s_j$ may be covered in the same iteration as some of the other elements $s_{j'}$ for $j' < j$. In this iteration, the greedy algorithm selected a set $S_i$ of minimum average cost; so this set $S_i$ has average cost at most that of $S_k$. It is the average cost of $S_i$ that gets assigned to $s_j$, and so we have

$$c_{s_j} = \frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}$$

We now simply add up these inequalities for all elements $s \in S_k$

$$\sum_{s \in S_k} c_s = \sum_{j=1}^{d} c_{s_j} \leq \sum_{j=1}^{d} \frac{w_k}{d - j + 1} = \frac{w_k}{d} + \frac{w_k}{d-1} + \ldots + \frac{w_k}{1} = H(d) \cdot w_k. \quad \blacksquare$$

# Set Cover greedy algorithm analysis

**(11.11)** *The set cover $\mathcal{C}$ selected by* `Greedy-Set-Cover` *has weight at most $H(d^*)$ times the optimal weight $w^*$.*

*Where* $d^* = \max_i |S_i|$ denotes the maximum size of any set.

Let $C*$ denote the optimum set cover, so that $w^* = \sum_{S_i \in \mathcal{C}^*} w_i$

For each of the sets in $C^*$, (11.10) implies $w_i \geq \frac{1}{H(d^*)} \sum_{s \in S_i} c_s$

Because these sets form a set cover, we have $\sum_{S_i \in \mathcal{C}^*} \sum_{s \in S_i} c_s \geq \sum_{s \in U} c_s$

Combining these with (11.9), we obtain the desired bound:

$$w^* = \sum_{S_i \in \mathcal{C}^*} w_i \geq \sum_{S_i \in \mathcal{C}^*} \frac{1}{H(d^*)} \sum_{s \in S_i} c_s \geq \frac{1}{H(d^*)} \sum_{s \in U} c_s = \frac{1}{H(d^*)} \sum_{S_i \in \mathcal{C}} w_i. \quad \blacksquare$$