Load Balancing

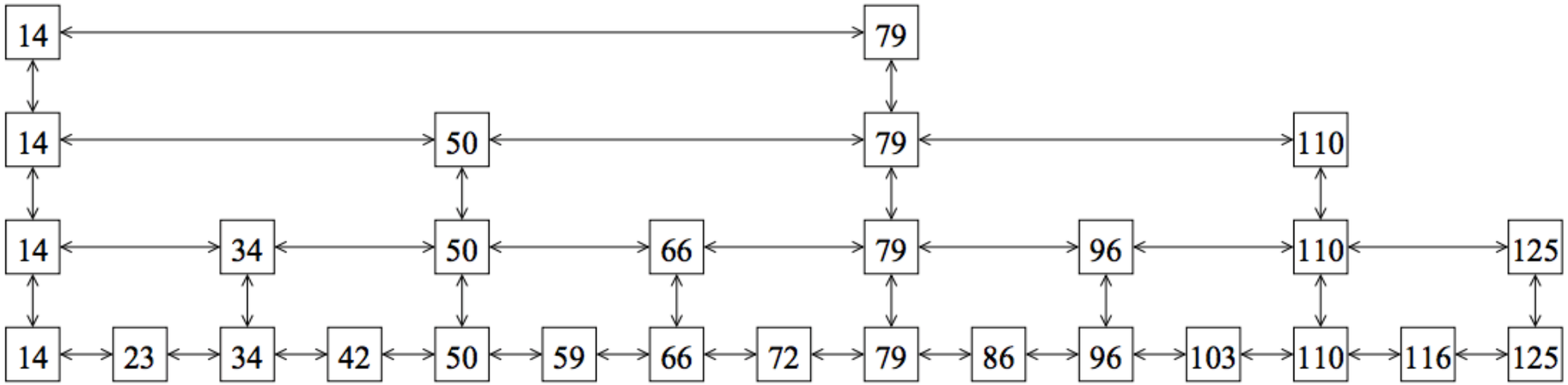
Admin

- Assignment 8 back soon

- Final review out tonight

Skip List Details

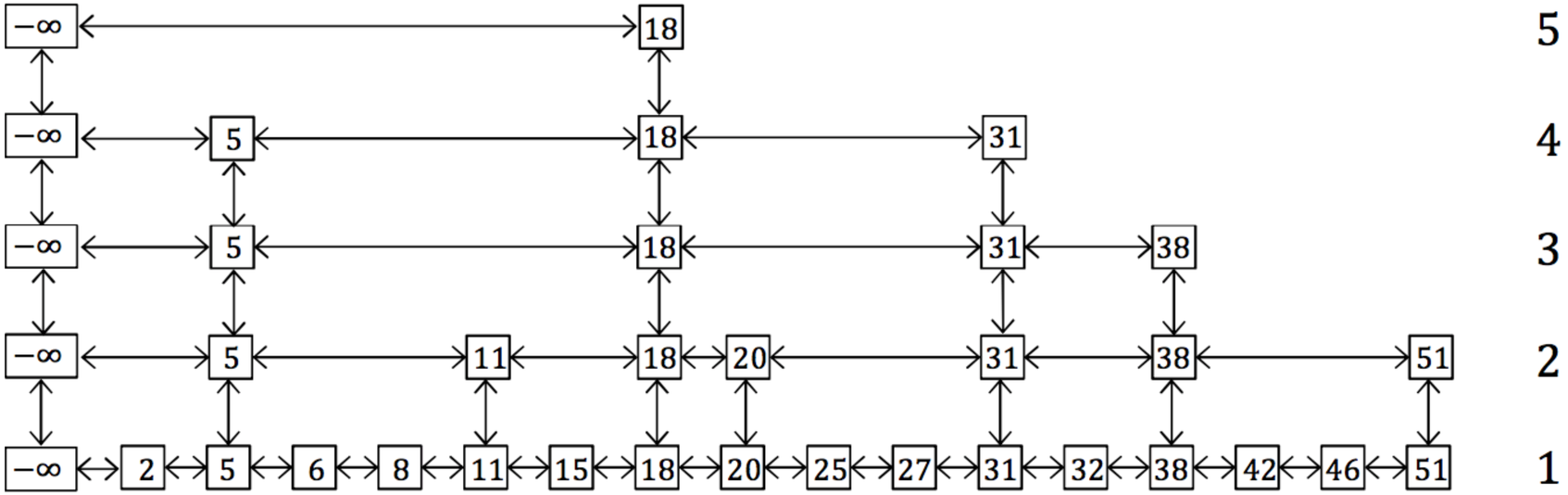
- Search(x):
  - Start at top list, go right just before value gets > target
  - Go down and repeat until element is found or hit bottom



- **Example:** Search for 72
  - Level 1: 14 too small, 79 too big; go down 14
  - Level 2: 14 too small, 50 too small, 79 too big; go down 50
  - Level 3: 50 too small, 66 too small, 79 too big; go down 66
  - Level 4: 66 too small, 72 spot on

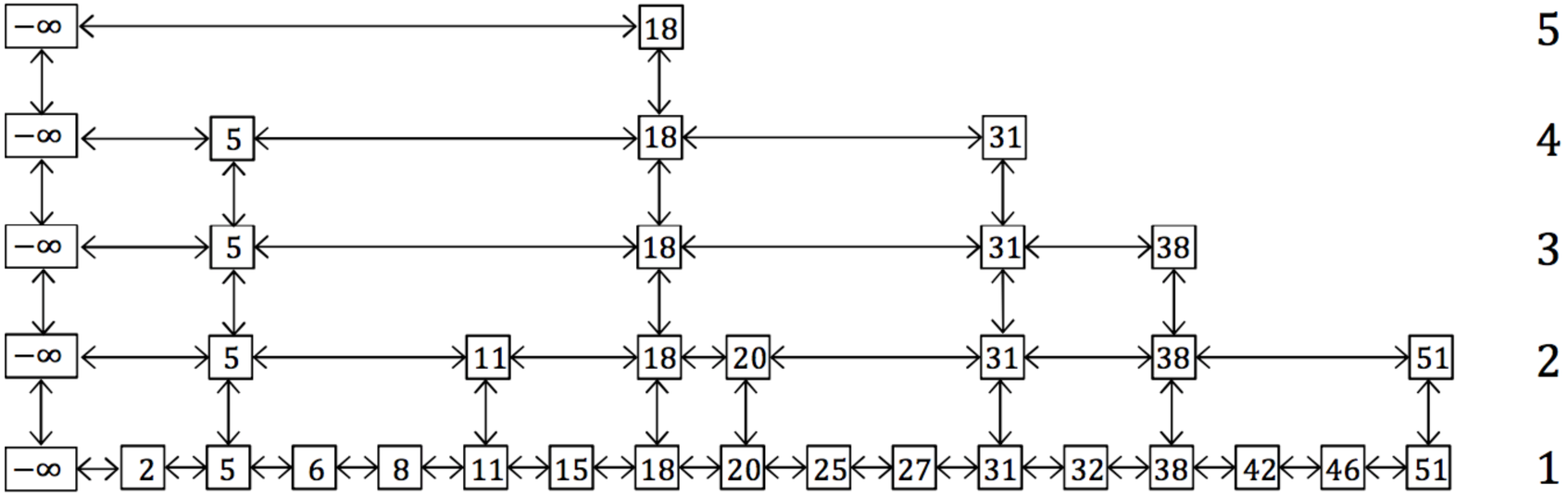
Skip List Analysis

- Let us first define the height of a skip list formally.
- Let L_k be the set of all items in level $k \geq 1$.
- Height of an element. $\ell(x) = \max\{k \mid x \in L_k\}$
- Height of a skip list. $h(L) = \max\{\ell(x) \mid x \in L_0\}$
-



Skip List Expected Analysis

- Expected height of a node:
  - $E[\ell(x)] = 1 + \frac{1}{2} \cdot 0 + \frac{1}{2}(1 + E[\ell(x)])$
  - $E[\ell(x)] = 2$
- Worst-case height? $h(L) = \max\{\ell(x) \mid x \in L\}$



Skip List Analysis

- **Claim.** A skip list with n elements has height $O(\log n)$ levels with high probability.
- Recall. (Informally) An event happens with high probability if the probability that it does not happen is polynomially small in n, that is, $\leq 1/n^c$ where the constant $c \geq 1$
- (More formally) Skip list of size n has $O(\log n)$ levels with high probability if the probability that it has more than $d \log n$ levels is at most $1/n^c$ where the constants c, d usually depend on each other
- **Proof idea.** What is the probability that an element gets promoted to level 1?
  - $1/2$

Skip List Analysis

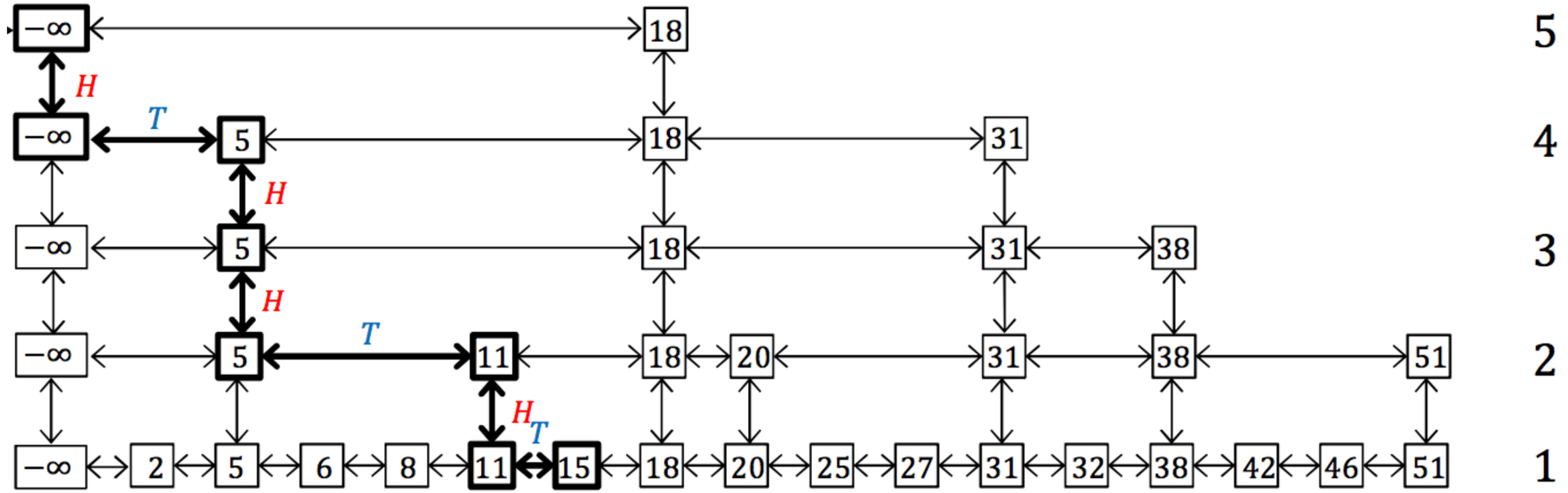
- **Claim.** A skip list with n elements has height $O(\log n)$ levels with high probability.
- **Proof.** For any $x \in L,\ k \geq 1$, the probability that height of x is k
- $\mathbf{Pr}[\ell(x) = k] = \frac{1}{2^k}$
- $\Pr[\ell(x) > k] = \sum_{k+1}^{\infty} \Pr[\ell(x) = i] = \sum_{i=k+1}^{\infty} \frac{1}{2^i} = \frac{1}{2^k}$
- $\Pr[h(L) > k] = \Pr[\cup_{x \in L} \ell(x) > k] \leq \sum_{x \in L} \Pr[\ell(x) > k] = \frac{n}{2^k}$

Union bound

- $\Pr[h(L) > c \log n] \leq \frac{1}{n^{c-1}}$ [pick any $c > 2$ for w.h.p.]
- Thus, height of skip is $O(\log n)$ with high probability

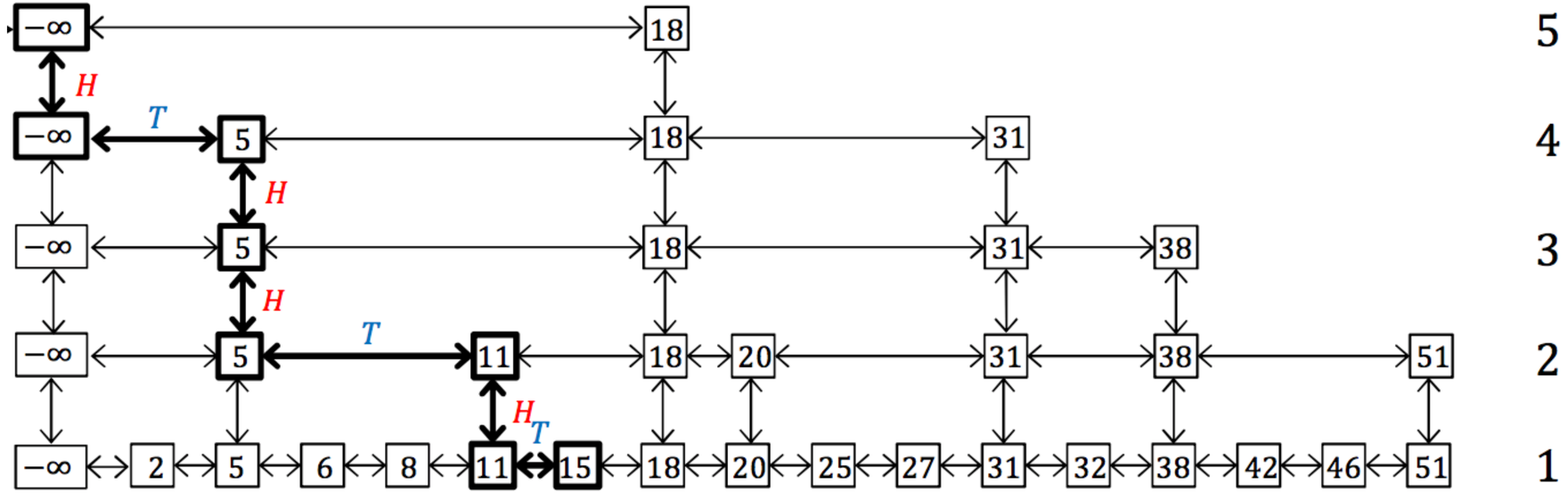
Skip List Search Cost

- **Claim.** Search cost in a skip list is $O(\log n)$ with high probability
- **Proof.**
- Idea think of the search path “backwards”
- Starting at the target element
- Going left or up until you reach top-left element



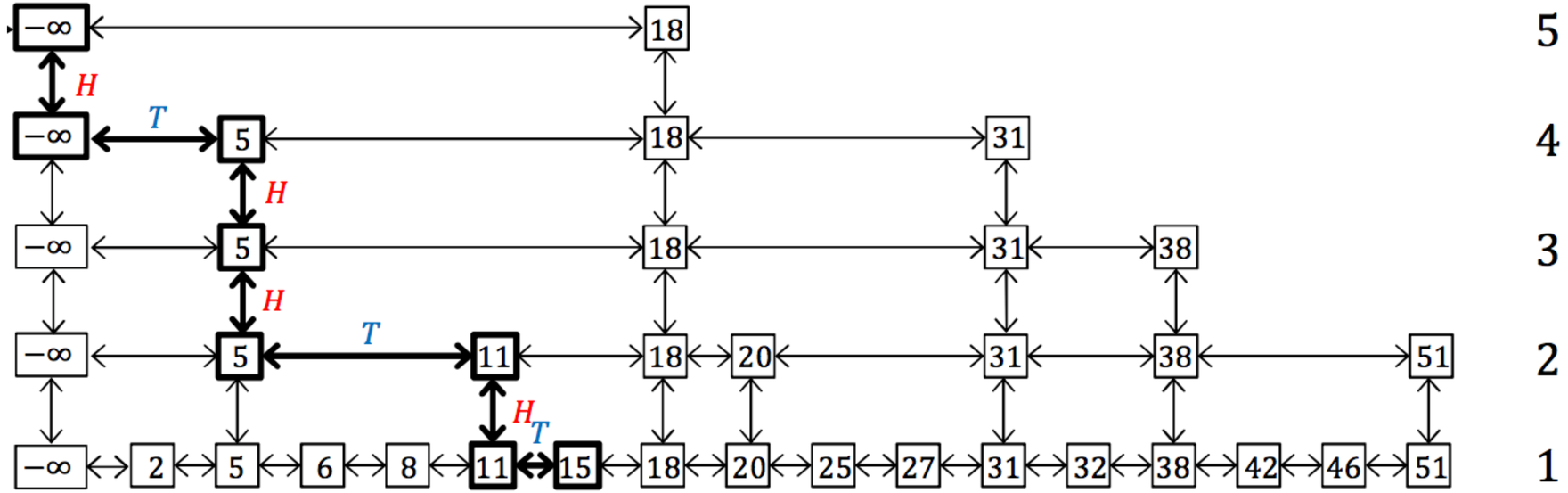
Skip List Search Cost

- Backwards search path, when do go up versus left?
- If node wasn't promoted (got tails here), then we go [came from] left
- If node was promoted (got heads here), then we go [came from] top
- How many consecutive tails in a row? (left moves on a level)
  - Same analysis as the height! $O(\log n)$
  - $O(\log^2 n)$ length overall—but I claimed $O(\log n)$ earlier



Skip List Search Cost

- We know height is $O(\log n)$ with high probability; say it is $c \log n$
- Thus, number of "up" moves is at most $c \log n$ with high probability
- Search path is a sequence of $HHHTTTHHTT\ldots$
- Search cost:
  - How many times do we need to flip a coin to get $c \log n$ heads with high probability?



Coin Flipping

- **Claim.** Number of flips until $c \log n$ heads is $\Theta(\log n)$ with high probability, that is, with probability $1 - 1/n^c$
- Note. Constant in $\Theta(\log n)$ will depend on c
- **Proof.**
  - Say we flip $10c \log n$ coins
  - When are there at least $c \log n$ heads?
  - Pr[exactly $c \log n$ heads]

$$= \binom{10c \log n}{c \log n} \cdot \left(\frac{1}{2}\right)^{c \log n} \cdot \left(\frac{1}{2}\right)^{9c \log n}$$

  - Pr[at most $c \log n$ heads] $\leq \binom{10c \log n}{c \log n} \cdot \left(\frac{1}{2}\right)^{9c \log n}$

Coin Flipping

- **Claim.** Number of flips until $c \log n$ heads is $\Theta(\log n)$ with high probability, that is, with probability $1 - 1/n^c$
- **Proof.**
  - Pr[at most $c \log n$ heads] $\leq \left(\frac{e \cdot 10c \log n}{c \log n}\right)^{c \log n} \cdot \left(\frac{1}{2}\right)^{9c \log n}$

$$= (10e)^{c \log n} \cdot \left(\frac{1}{2}\right)^{9c \log n}$$

$$= 2^{\log(10e) \cdot c \log n} \cdot \left(\frac{1}{2}\right)^{9c \log n}$$

$$= 2^{(\log(10e)-9) \cdot c \log n} = 2^{-d \log n}$$

$$= 1/n^d$$

- If we instead look at probability of at most $d'c \log n$ heads, as $d' \to \infty$, $d = 9 - \log(10e) \to \infty$, independent of c

Skip Lists

- Using $O(\log n)$ linked lists, achieve same performance as binary search tree
- No stored information about balance, no tricky balancing rules
- Just flip coins while inserting each new element to decide what lists it goes in

Approximation Algorithms

Aside: Online Algorithms

- For algorithms we've seen, we have all data up front
- Not always true in practice!
- What happens when data comes in gradually, and you need to commit to a solution before you see all of it?

Challenges: Approximation Algorithms

- Approximating problems that are NP hard
  - Main challenge is showing that the algorithm performs close to optimal when the optimal solution is not known/NP hard
  - Usually done by lower (upper) bounding the cost of the optimal solution for minimization (maximization) problems

- Approximation for online algorithms
  - High benchmark. Comparison against an optimal that knows the entire future, while the algorithm does not even know the next element
  - k-competitive: if the optimal *offline* algorithm has cost OPT, our algorithm has cost $k \cdot OPT$

Online: Ski Rental Problem

- Assume that you are taking ski lessons
- At some point (after t days) the ski season ends. But you don't have any information about when that will be.
- Question: rent or buy the skis?
- Cost of renting $\$1$
- Cost of buying $\$B$
- **Offline strategy.** If you knew in advance when the ski season ends (you know t) what is the best strategy?
  - If $t \geq B$ times, then buy, else rent
  - In other words, optimal offline cost is $\min\{t, B\}$

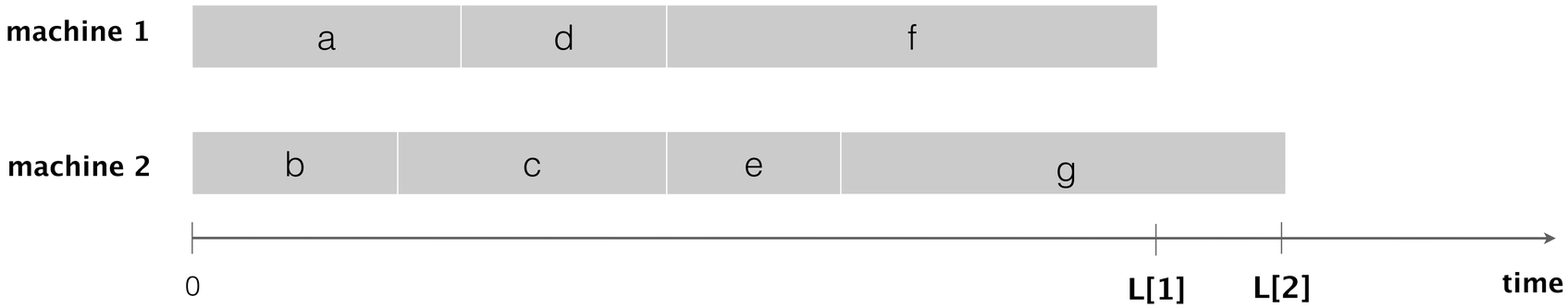
Online: Ski Rental Problem

- **Online strategy.** We need to figure out a decision point, a number k such you buy skis on the kth visit (renting before then)
- **Claim.** If we set $k = B$ (the cost of buying skis), we are gauranteed to never pay more than twice of the best offline optimal strategy. That is, buying on the Bth ski visit is $2-$competitive
- Offline cost is $\min\{t, B\} = B$
- Online strategy's cost?
  - If $t \leq B$, then our cost is t, and OPT has cost t. In that case, we are 2-competitive (in fact we are 1-competitive)
  - If $t > B$, then our cost is $2B$, and OPT has cost B. In that case, we are 2-competitive

Load Balancing

Load Balancing

- **Input.** m identical machines
- n jobs with processing times $t_1, \ldots, t_m$, where job j has processing time t_j (on any machine)
- Job j must run contiguously on one machine
- A machine can process at most one job at a time.
- Let $S[i]$ be the subset of jobs assigned to machine i.
- The **load of machine** i is $L[i] = \sum_{j \in S[i]} t_j$ (total processing time).



Load Balancing

- The makespan of an algorithm is the maximum load on any machine
  $L = \max_i L[i]$
- Load balancing Problem. Assign jobs to machines so as to minimize makespan.
- **Claim.** Load balancing is NP hard even with $m = 2$ machines
- **Proof.** Reduction from PARTITION problem.
- We will design an approximation algorithm for this problem
- [Greedy returns!] Consider the following greedy strategy:
  - Fix some order on the jobs
  - Assign job j to machine i whose load is smallest so far

Load Balancing: Greedy

- Go through the jobs one by one
- Assign each job to the machine with the smallest load so far

- How can we keep track of this efficiently?
    - Priority queue

Load Balancing: Greedy

LIST-SCHEDULING $(m, n, t_1, t_2, \ldots, t_n)$

FOR $i = 1$ TO m

$L[i] \leftarrow 0.$ ← load on machine i

$S[i] \leftarrow \emptyset.$ ← jobs assigned to machine i

FOR $j = 1$ TO n

$i \leftarrow \text{argmin}_k L[k].$ ← machine i has smallest load

$S[i] \leftarrow S[i] \cup \{ j \}.$ ← assign job j to machine i

$L[i] \leftarrow L[i] + t_j.$ ← update load of machine i

RETURN $S[1], S[2], \ldots, S[m].$

- Running time?
  - $O(n \log m)$ using a priority queue for loads $L[k]$

Load Balancing: Greedy Analysis

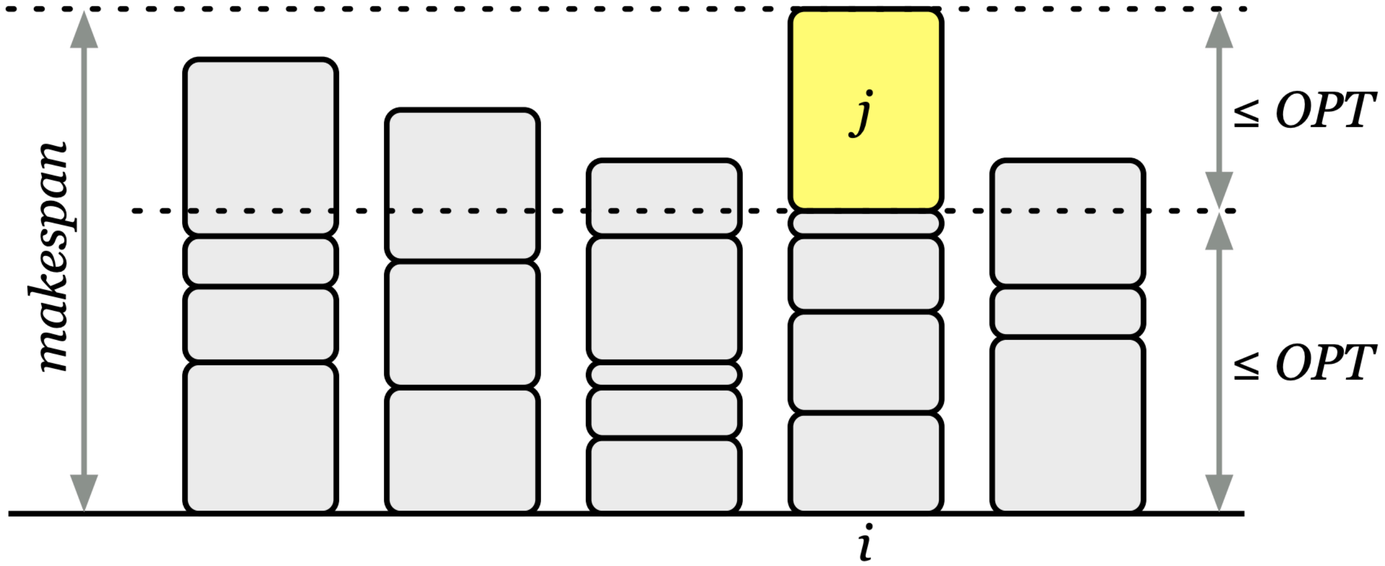
- **Claim.** Greedy algorithm is a 2-approximation.
- To show this, we need to show greedy solution never more than a factor two worse than the optimal
- **Challenge.** We don't know the optimal solution. In fact, finding the optimal is NP hard.
- We want to:
  - Lower bound the cost of optimal solution
  - A good enough lower bound can help show that our algorithm cannot be too much worse than the optimal
- In our problem, what are some lower bounds on the makespan of even an optimal algorithm?

Load Balancing: Greedy Analysis

- Let OPT be the optimal makespan.
- **Lemma.** OPT $\geq \max_j t_j$.
- **Proof.** Some machine must process the most time-consuming job.
- Any other lower bounds?
- **Lemma.** OPT $\geq \frac{1}{m} \sum_j t_j$
- **Proof.**
  - The total processing time is $\sum_j t_j$
  - Some machine must do a $1/m$ fraction of the total work.

Greedy is a 2-Approximation

- **Proof.** Consider load $L[i]$ of bottleneck machine i ← machine that ends up with highest load
  - Let j be the last scheduled job on machine i
  - When job j was assigned to machine i, i must have had the smallest load
  - That is, $L[i] - t_j \leq L[k] \quad \forall 1 \leq k \leq m$



Greedy is a 2-Approximation

- **Proof.** Consider load $L[i]$ of bottleneck machine i
  - Let j be the last scheduled job on machine i
  - When job j was assigned to machine i, i must have had the smallest load
  - That is, $L[i] - t_j \leq L[k] \quad \forall 1 \leq k \leq m$
  - Summing over all k and diving by m

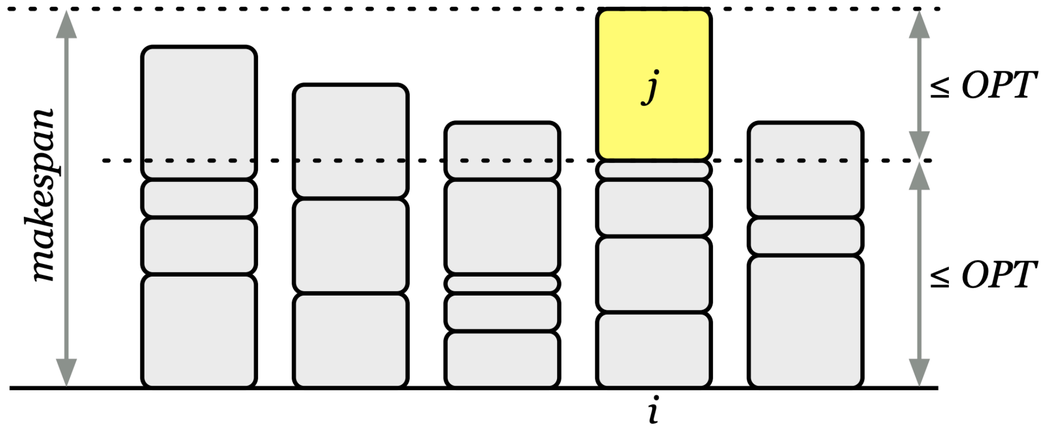
$$
\begin{aligned}
L[i] - t_j &\leq \frac{1}{m} \sum_k L[k] \\
&\leq \frac{1}{m} \sum_{j'} t_{j'} \\
&\leq \text{OPT}
\end{aligned}
$$

Greedy is a 2-Approximation

- **Proof.**
- Consider load $L(i)$ of bottleneck machine i
- $L[i] - t_j \leq$ OPT

- We know that $t_j \leq$ OPT
- Thus, $L = L[i] \leq \text{OPT} + t_j$

$\leq 2\text{OPT}$ ∎



Greedy is a 2-Approximation

- Is our analysis tight?
- Close to it.
- Consider $m(m-1)$ jobs of length 1 and 1 job of length m
- How would greedy schedule these jobs?
  - Greedy will evenly divide the first $m(m-1)$ jobs among m machines, will place the final long job on any one machine
  - Makespan: $m-1+m=2m-1$
- How would optimal schedule it?
  - Give the long job to one machine, the rest split the other small jobs with a makespan m
- Ratio: $(2m-1)/m \approx 2$

Greedy is Online

- Notice that our greedy algorithm is an online algorithm
- Assigns jobs to machines in the order they arrive
    - Does not depend on future jobs
- Can we do better, if we assume all jobs are available at start time?
- **Offline.** Slight modification of greedy gets better approximation!

Improving on Online Greedy

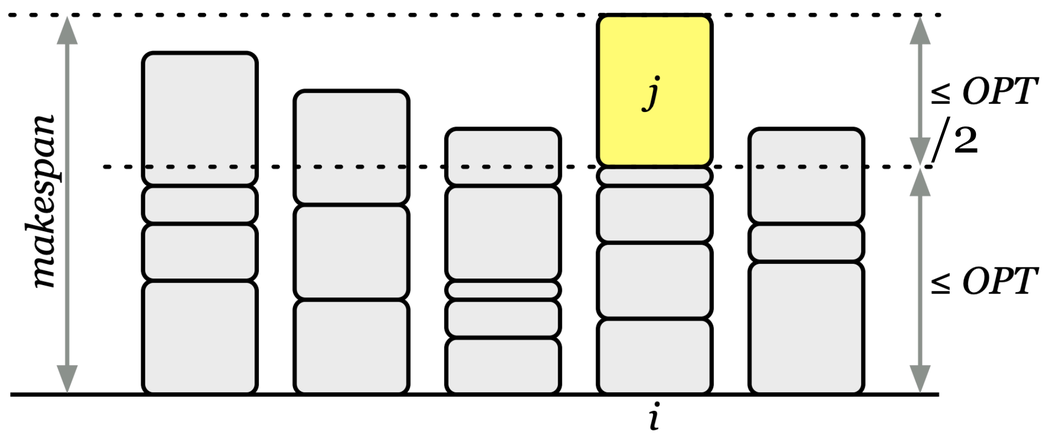
- Worst case of our greedy algorithm: spreading jobs out evenly when a giant job at the end messed things up
- What can we do to avoid this?
  - Idea: deal with larger jobs first
  - Small jobs can only hurt so much
- Turns out this improves our approximation factor
- **Longest-processing-time (LPT) first**. Sort n jobs in decreasing order of processing times; then run the greedy algorithm on them
- **Claim.** LPT has a makespan at most $1.5 \cdot$ OPT
- **Observation.** If we have fewer than m jobs, then the greedy solution is clearly optimal (as it puts each job on its own machine)

LPT-first is a 1.5-Approximation

- **Lemma.** LPT-first has a makespan at most $1.5 \cdot$ OPT
- **Observation.**
  - If we have fewer than m jobs, then the greedy solution is clearly optimal (as it puts each job on its own machine)
- **Claim.** If more than m jobs then, OPT $\geq 2 \cdot t_{m+1}$
- **Proof.** Consider the first $m + 1$ jobs in sorted order.
  - They each take at least t_{m+1} time
  - $m + 1$ jobs and m machines, there must be a machine with at least two jobs
  - Thus the optimal makespan OPT $\geq 2 \cdot t_{m+1}$

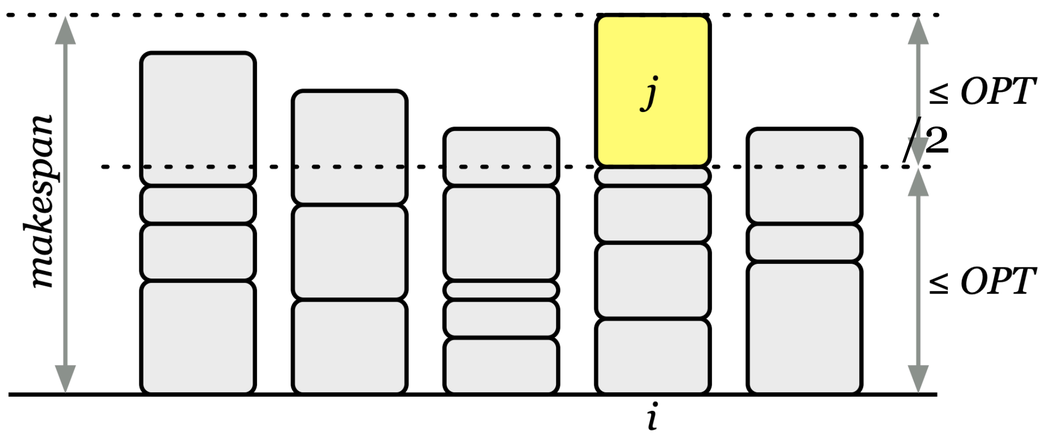
LPT-first is a 1.5-Approximation

- **Lemma.** LPT-first has a makespan at most $1.5 \cdot$ OPT
- **Proof.** Similar to our original proof. Consider the machine M_i that has the maximum load
- If M_i has a single job, then our algorithm is optimal
- Suppose M_i has at least two jobs and let t_j be the last job assigned to the machine, note that $j \geq m + 1$ (why?)
- Thus, $t_j \leq t_{m+1} \leq \frac{1}{2}$OPT



LPT-first is a 1.5-Approximation

- **Lemma.** LPT-first has a makespan at most $1.5 \cdot \text{OPT}$
- **Proof.** Similar to our original proof. Consider the machine M_i that has the maximum load
- If M_i has a single job, then our algorithm is optimal
- Suppose M_i has at least two jobs and let t_j be the last job assigned to the machine, note that $j \geq m + 1$ (why?)
- Thus, $t_j \leq t_{m+1} \leq \frac{1}{2}\text{OPT}$
- $L[i] - t_j \leq \text{OPT}$
- $L[i] \leq \frac{3}{2}\text{OPT}$ ∎



Is our 1.5-Approximation tight?

- **Question.** Is out $3/2$-approximation analysis tight?
  - Turns out, no
- **Theorem [Graham 1969].** LPT-first is a $4/3$-approximation.
  - Proof via a more sophisticated analysis of the same algorithm
- **Question.** Is the $4/3$-approximation analysis tight?
  - Pretty much.
- Example
  - m machines, $n = 2m + 1$ jobs
  - 2 jobs each of length $m, m+1, \ldots, 2m-1$ + one job of length m
  - Approximation ratio $= (4m-1)/3m \approx 4/3$

Can we do better than $4/3$?

- Long series of improvements
- Polynomial time algorithm for *any* constant approximation [Hochbaum Shmoys 87]
- Specifically: $(1+\epsilon)$ approximation in $O\left((n/\epsilon)^{1/\epsilon^2}\right)$ time
- **PTAS:** Polynomial time approximation scheme
- For any desired constant-factor approximation, there exists a polynomial-time algorithm

Acknowledgments

- Some of the material in these slides are taken from
  - Kleinberg Tardos Slides by Kevin Wayne (https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf)
  - Jeff Erickson's Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)
  - Lecture slides: https://web.stanford.edu/class/archive/cs/cs161/cs161.1138/