CSC2420: Algorithm Design, Analysis and Theory Spring (aka Winter) 2015

Allan Borodin and Lalla Mouatadid (TA)

January 8, 2015

Lecture 1

Course Organization:

1. **Sources**: No one text; lots of sources including specialized graduate textbooks, my posted lecture notes (beware typos), lecture notes from other Universities, and papers. Very active field. Foundational course but we will discuss some recent work and research problems.
2. **Lectures and Tutorials**: One two hour lecture per week with tutorials as needed and requested; TA is Lalla Mouatadid.
3. **Grading**: Will depend on how many students are taking this course for credit. In previous offerings there were three assignments with an occasional opportunity for some research questions. I may have to have some more supervised aspect to the grading.
4. **Office hours**: I will be posting regular office hours for my CSC 200 course but mainly when I am in (which is most of the time), my door is open and I welcome questions (unless I am preoccupied). So feel free to drop by and/or email me to schedule a time. My office is SF 2303B and my email is bor@cs.toronto.edu. The course web page is www.cs.toronto.edu/~bor/2420s15

What is appropriate background?

- In short, a course like our undergraduate CSC 373 is essentially the prerequisite.
- Any of the popular undergraduate texts. For example, Kleinberg and Tardos; Cormen, Leiserson, Rivest and Stein; DasGupta, Papadimitriou and Vazirani.
- It certainly helps to have a good math background and in particular understand basic probability concepts, and some graph theory.

BUT any CS/ECE/Math graduate student (or mathematically oriented undergrad) should find the course accessible and useful.

Comments and disclaimers on the course perspective

- This is a graduate level "foundational course". However, I will focus somewhat on my current research perspective; this then does not represent a standard introduction to the field.
- But in my defense, perhaps most graduate algorithms courses are biased towards some research perspective. I do not think there is a standard course in the same way that the previously mentioned texts represent a standard for an undergraduate course.
- Given that CS might be considered (to some extent) The Science and Engineering of Algorithms, one cannot expect any comprehensive introduction to algorithm design and analysis. Even within theoretical CS, there are many focused courses and texts for particular subfields; e.g. randomized algorithms, approximation algorithms, linear programming (and more generally mathematical programming), online algorithms, parallel algorithms, streaming algorithms, sublinear time algorithms.
- I have added the word theory to the course title to reflect my interest in making generally informal concepts a little more precise.

Reviewing some basic algorithmic paradigms

We begin with some "conceptually simple search/optimization algorithms".

The conceptually simplest "combinatorial" algorithms

Given an optimization problem, it seems to me that the conceptually simplest approaches are:

- brute force search
- greedy
- local search

Comment

- We usually dismiss brute force as it really isn't much of an algorithm approach but might work for small enough problems.
- Moreover, sometimes we can combine some aspect of brute force search with another approach as we will see by combining brute force and greedy.

Greedy algorithms in CSC373

Some of the greedy algorithms we study in different offerings of CSC 373

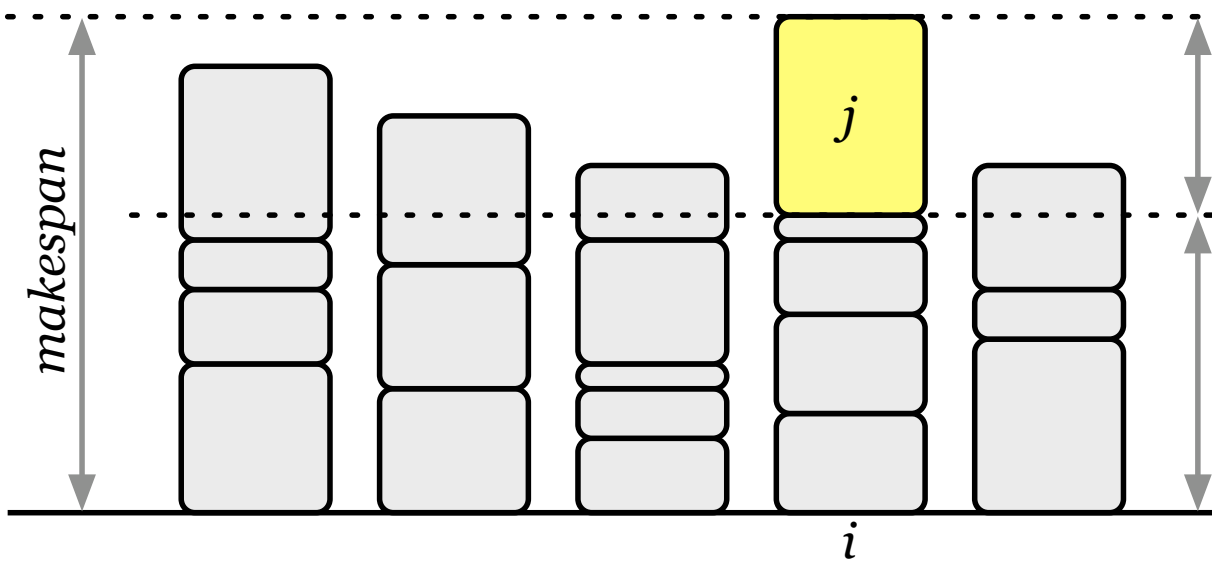
- The optimal algorithm for the fractional knapsack problem and the approximate algorithm for the proportional profit knapsack problem.
- The optimal unit profit interval scheduling algorithm and 3-approximation algorithm for proportional profit interval scheduling.
- The 2-approximate algorithm for the unweighted job interval scheduling problem and similar approximation for unweighted throughput maximization.
- Kruskal and Prim optimal algorithms for minimum spanning tree.
- Huffman's algorithm for optimal prefix codes.
- Graham's online and LPT approximation algorithms for makespan minimization on identical machines.
- Maximal matching approximation for unweighted vertex cover.
- The "natural greedy" algorithm for set cover.

Greedy algorithms: Graham's online and LPT makespan algorithms

- Let's start with these two greedy algorithms that date back to 1966 and 1969 technical reports.
- These are good starting points since (preceding NP-completeness) Graham conjectured that these are hard (requiring exponential time) problems to compute optimally but for which there were worst case approximation ratios (although he didn't use that terminology).
- This might then be called the start of worst case approximation algorithms. Moreover, there are some general concepts to be observed in this work and even after nearly 50 years still some open questions concerning such makespan problems.

The makespan problem

- The input consists of n jobs $\mathcal{J} = J_1 \ldots, J_n$ that are to be scheduled on m identical machines.
- Each job J_k is described by a processing time (or load) p_k.
- The goal is to minimize the latest finishing time (maximum load) over all machines.
- That is, the goal is a mapping $\sigma : \{1, \ldots, n\} \to \{1, \ldots, m\}$ that minimizes $\max_k \left(\sum_{\ell:\sigma(\ell)=k} p_\ell \right)$.

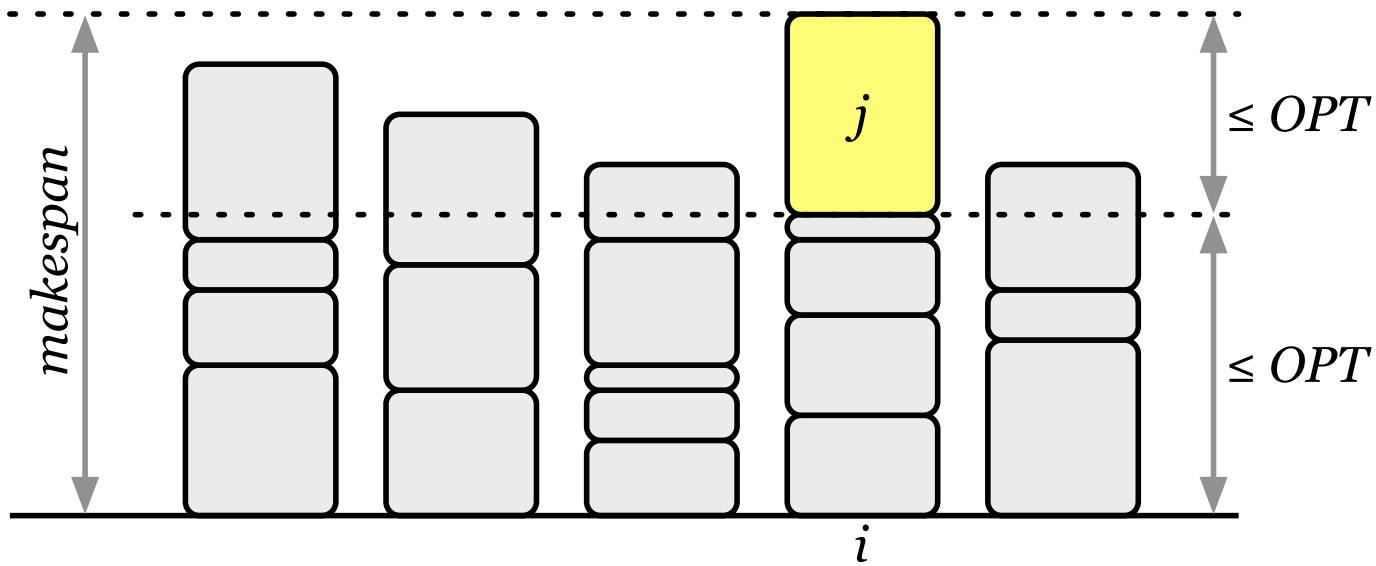


[picture taken from Jeff Erickson's lecture notes]

Graham's online greedy algorithm

Consider input jobs in any order (e.g. as they arrive in an *online* setting) and schedule each job J_j on any machine having the least load thus far.

- We will see that the **approximation ratio** for this algorithm is $2 - \frac{1}{m}$; that is, for any set of jobs $\mathcal{J}$, $C_{Greedy}(\mathcal{J}) \leq (2 - \frac{1}{m}) C_{OPT}(\mathcal{J})$.
  - C_A denotes the cost (or makespan) of a schedule A.
  - OPT stands for any optimum schedule.
- **Basic proof idea:** $OPT (\geq \sum_j p_j)/m$; $OPT \geq max_j p_j$



[picture taken from Jeff Erickson's lecture notes]

Graham's online greedy algorithm

Consider input jobs in any order (e.g. as they arrive in an online setting) and schedule each job J_j on any machine having the least load thus far.

- In the online "competitive analysis" literature the ratio $\frac{C_A}{C_{OPT}}$ is called the **competitive ratio** and it allows for this ratio to just hold in the limit as C_{OPT} increases. This is the analogy of *asymptotic approximation ratios*.

NOTE: Often, I will not provide proofs in the lecture notes but rather will do or sketch proofs in class (or leave proof as an exercise).

- The approximation ratio for the online greedy is "tight" in that there is a sequence of jobs forcing this ratio.
- This bad input sequence suggests a better algorithm, namely the LPT (offline or sometimes called semi-online) algorithm.

Graham's LPT algorithm

Sort the jobs so that $p_1 \geq p_2 \ldots \geq p_n$ and then greedily schedule jobs on the least loaded machine.

- The (tight) approximation ratio of LPT is $\left(\frac{4}{3} - \frac{1}{3m}\right)$.
- It is believed that this is the best "greedy" algorithm but how would one prove such a result? This of course raises the question as to what is a greedy algorithm.
- We will present the priority model for greedy (and greedy-like) algorithms. I claim that all the algorithms mentioned on slide 6 can be formulated within the priority model.
- Asssuming we maintain a priority queue for the least loaded machine,
  - the online greedy algorithm would have time complexity $O(n \log m)$ which is $(n \log n)$ since we can assume $n \geq m$.
  - the LPT algorithm would have time complexity $O(n \log n)$.

Partial Enumeration Greedy

- Combining the LPT idea with a brute force approach improves the approximation ratio but at a significant increase in time complexity.
- I call such an algorithm a "partial enumeration greedy" algorithm.

Optimally schedule the largest k jobs (for $0 \leq k \leq n$) and then greedily schedule the remaining jobs (in any order).

- The algorithm has approximation ratio no worse than $\left(1 + \frac{1-\frac{1}{m}}{1+\lfloor k/m \rfloor}\right)$.
- Graham also shows that this bound is tight for $k \equiv 0 \mod m$.
- The running time is $O(m^k + n \log n)$.
- Setting $k = \frac{1-\epsilon}{\epsilon} m$ gives a ratio of at most $(1 + \epsilon)$ so that *for any fixed* m, this is a PTAS (polynomial time approximation scheme). with time $O(m^{m/\epsilon} + n \log n)$.

Makespan: Some additional comments

- There are many refinements and variants of the makespan problem.
- There was significant interest in the best competitive ratio (in the online setting) that can be achieved for the makespan problem.
- The online greedy gives the best online ratio for m $= 2,3$ but better bounds are known for $m \geq 4$.
  Basic idea: leave some room for a possible large job; this forces the online algorithm to be non-greedy in some sense but still within the priority model which subsumes online algorithms.
- Randomization can provide somewhat better competitive ratios.
- Makespan has been actively studied with respect to three other machine models.

The uniformly related machine model

- Each machine i has a speed s_i
- Recall that each job J_j is described by a processing time or load p_j.
- The processing time to schedule job J_j on machine i is p_j/s_i.
- There is an online algorithm that achieves a constant competitive ratio.
- I think the best known online ratio is 5.828 due to Berman et al following the first constant ratio by Aspnes et al.
- Ebenlendr and Sgall establish an online inapproximation of 2.564 following the 2.428 inapproximation of Berman et al.

The restricted machines model

- Every job J_j is described by a pair (p_j, S_j) where $S_j \subseteq \{1, \ldots, m\}$ is the set of machines on which J_j can be scheduled.
- This (and the next model) have been the focus of a number of papers (for both online and offline) and there has been some relatively recent progress in the offline restricted machines case.
- Even for the case of two allowable machines per job (i.e. the *graph orientation problem*), this is an interesting problem and we may look at some recent work later.
- Azar et al show that $\log_2(m)$ (resp. $\ln(m)$) is (up to ± 1) the best competitive ratio for deterministic (resp. randomized) online algorithms with the upper bounds obtained by the natural greedy algorithm.
- It is not known if there is an offline greedy-like algorithm for this problem that achieves a constant approximation ratio. Regev [IPL 2002] shows an $\Omega(\frac{\log m}{\log \log m})$ inapproximation for fixed order priority algorithms for the restricted case when every job has 2 allowable machines.

The unrelated machines model

- The most general of the machine models.
- Now a job J_j is represented by a vector $(p_{j,1}, \ldots, p_{j,m})$ where $p_{j,i}$ is the time to process job J_j on machine i.
- A classic result of Lenstra, Shmoys and Tardos [1990] shows how to solve the (offline) makespan problem in the unrelated machine model with approximation ratio 2 using LP rounding.
- There is an online algorithm with approximation $O(\log m)$. Currently, this is the best approximation known for greedy-like (e.g. priority) algorithms even for the restricted machines model although there has been some progress made in this regard (which we will discuss later).
- NOTE: All statements about what we will do later should be understood as intentions and not promises.

The knapsack problem

The knapsack problem

- Input: Knapsack size capacity C and n items $\mathcal{I} = \{I_1, \ldots, I_n\}$ where $I_j = (v_j, s_j)$ with v_j (resp. s_j) the profit value (resp. size) of item I_j.
- Output: A feasible subset $S \subseteq \{1, \ldots, n\}$ satsifying $\sum_{j \in S} s_j \leq C$ so as to maximize $V(S) = \sum_{j \in S} v_j$.

Note: I would prefer to use approximation ratios $r \geq 1$ (so that we can talk unambiguously about upper and lower bounds on the ratio) but many people use approximation ratios $\rho \leq 1$ for maximization problems; i.e. $ALG \geq \rho OPT$. For certain topics, this is the convention.

- It is easy to see that the most natural greedy methods (sort by non-increasing profit densities $\frac{v_j}{s_j}$, sort by non-increasing profits v_j, sort by non-decreasing size s_j) will not yield any constant ratio.

The partial enumeration greedy PTAS for knapsack

$PGreedy_k$ Algorithm

Sort $\mathcal{I}$ so that $\frac{v_1}{s_1} \geq \frac{v_2}{s_2} \ldots \geq \frac{v_n}{s_n}$
For every feasible subset $H \subseteq \mathcal{I}$ with $|H| \leq k$
    Let $R = \mathcal{I} - H$ and let $S_H := H$
    Consider items in R (in the order of profit densities)
    and greedily add items to S_H that do not exceed knapsack capacity C.
        % It is sufficient for the approximation ratio to stop
            as soon as an item is too large to fit
End For
Output: the S_H having maximum profit.

Sahni's PTAS result

Theorem (Sahni 1975): $V(OPT) \leq (1 + \frac{1}{k})V(PGreedy_k)$.

- This algorithm takes time kn^k and setting $k = \frac{1}{\epsilon}$ yields a $(1 + \epsilon)$ approximation running in time $\frac{1}{\epsilon} n^{\frac{1}{\epsilon}}$.
- An *FPTAS* is an algorithm achieving a $(1 + \epsilon)$ approximation with running time $poly(n, \frac{1}{\epsilon})$. There is an FPTAS for the knapsack problem (using dynamic programming and scaling the input values) so that the PTAS algorithm was quickly subsumed but still the partial enumeration technique is useful in other settings.
- In particular, more recently this technique (for $k = 3$) was used to achieve an $\frac{e}{e-1} \approx 1.58$ approximation for monotone submodular maximization subject to a knapsack constraint.
- It is NP-hard to do better than a $\frac{e}{e-1}$ approximation for submodular maximization subject to a cardinality constraint and hence this is also the best possible ratio for submodular maximization subject to a knapsack constraint.

The priority algorithm model and variants

Before temporarily leaving greedy (and greedy-like) algorithms, I want to present the priority algorithm model and how it can be extended in (conceptually) simple ways to go beyond the power of the priority model.

- What is the intuitive nature of a greedy algorithm as exemplified by the CSC 373 algorithms mentioned last class)? With the exception of Huffman coding (which we can also deal with) all these algorithms consider one input item in each iteration and make an irrevocable "greedy" decision about that item..
- We are then already assuming that the class of search/optimization problems we are dealing with can be viewed as making a decision D_k about each input item I_k (e.g. on what machine to schedule job I_k in the makespan case) such that $\{(I_1, D_1), \ldots, (I_n, D_n)\}$ constitutes a feasible solution.

Priority model continued

- Note: that a problem is only fully specified when we say how input items are represented.
- We mentioned that a "non-greedy" online algorithm for identical machine makespan can improve the competitive ratio; that is, the algorithm does not always place a job on the (or a) least loaded machine (i.e. does not make a greedy or locally optimal decision in each iteration). It isn't always obvious if or how to define a "greedy" decision but for many problems the definition of greedy can be informally phrased as "live for today" (i.e. assume the current input item could be the last item) so that the decision should be an optimal decision given the current state of the computation.

Greedy decisions and priority algorithms continued

- For example, in the knapsack problem, a greedy decision always takes an input if it fits within the knapsack constraint and in the makespan problem, a greedy decision always schedules a job on some machine so as to minimize the increase in the makespan. (This is somewhat more general than saying it must place the item on the least loaded machine.)
- If we do not insist on greediness, then priority algorithms might best have been called myopic algorithms.
- We have both fixed order priority algorithms (e.g. unweighted interval scheduling and LPT makespan) and adaptive order priority algorithms (e.g. the set cover greedy algorithm and Prim's MST algorithm).
- The key concept is to indicate how the algorithm chooses the order in which input items are considered. We cannot allow the algorithm to choose say "an optimal ordering".
- We might be tempted to say that the ordering has to be determined in polynomial time but that gets us into the "tarpit" of trying to prove what can and can't be done in (say) polynomial time.

Informal definition of a priority algorithm

- We take an information theoretic viewpoint in defining the orderings we allow.
- Lets first consider fixed priority algorithms. Since I am using this framework mainly to argue negative results (e.g. a priority algorithm for the given problem cannot achieve a stated approximation ratio), we will view the semantics of the model as a game between the algorithm and an adversary.
- Initially there is some (possibly infinite) set $\mathcal{J}$ of potential inputs. The algorithm chooses a total ordering π on $\mathcal{J}$. Then the adversary selects a subset $\mathcal{I} \subset \mathcal{J}$ of actual inputs so that $\mathcal{I}$ becomes the input to the priority algorithm. The input items $I_1, \ldots, I_n$ are ordered according to π.
- In iteration k for $1 \leq k \leq n$, the algorithm considers input item I_k and based on this input and all previous inputs and decisions (i.e. based on the current state of the computation) the algorithm makes an irrevocable decision D_k about this input item.

The fixed (order) priority algorithm template

$\mathcal{J}$ is the set of all possible input items
Decide on a total ordering π of $\mathcal{J}$
Let $\mathcal{I} \subset \mathcal{J}$ be the input instance
$S := \varnothing$ % S is the set of items already seen
$i := 0$ % $i = |S|$
while $\mathcal{I} \setminus S \neq \varnothing$ **do**
    $i := i + 1$
    $\mathcal{I} := \mathcal{I} \setminus S$
    $I_i := \min_\pi \{I \in \mathcal{I}\}$
    make an irrevocable decision D_i concerning I_i
    $S := S \cup \{I_i\}$
end

Figure : The template for a fixed priority algorithm

Some comments on the priority model

- A special (but usual) case is that π is determined by a function $f : \mathcal{J} \to \Re$ and and then ordering the set of actual input items by increasing (or decreasing) values $f()$. (We can break ties by say using the index of the item to provide a total ordering of the input set.) N.B. We make no assumption on the complexity or even the computability of the ordering π or function f.
- NOTE: Online algorithms are fixed order priority algorithms where the ordering is given *adversarially*; that is, the items are ordered by the index of the item.
- As stated we do not give the algorithm any additional information other than what it can learn as it gradually sees the input sequence.
- However, we can allow priority algorithms to be given some (hopefully easily computed) global information such as the number of input items, or say in the case of the makespan problem the minimum and/or maximium processing time/load of any input item. (Some inapproximation results can be easily modified to allow such global information.)

The adaptive priority model template

$\mathcal{J}$ is the set of all possible input items
$\mathcal{I}$ is the input instance
$S := \varnothing$ % S is the set of items already considered
$i := 0$ % $i = |S|$
while $\mathcal{I} \setminus S \neq \varnothing$ **do**
    $i := i + 1$
    decide on a total ordering π_i of $\mathcal{J}$
    $\mathcal{I} := \mathcal{I} \setminus S$
    $I_i := \min_{\leq_{\pi_i}} \{I \in \mathcal{I}\}$
    make an irrevocable decision D_i concerning I_i
    $S := S \cup \{I_i\}$
    $\mathcal{J} := \mathcal{J} \setminus \{I : I \leq_{\pi_i} I_i\}$
    % some items cannot be in input set
end

Figure : The template for an adaptive priority algorithm

Inapproximations with respect to the priority model

Once we have a precise model, we can then argue that certain approximation bounds are not possible within this model. Such inapproximation results have been established with respect to priority algorithms for a number of problems but for some problems much better approximations can be established using extensions of the model.

1. For the weighted interval selection (a *packing problem*) with arbitrary weighted values (resp. for proportinal weights $v_j = |f_j - s_j|$), no priority algorithm can achieve a constant approximation (respectively, better than a 3-approximation).
2. For the knapsack problem, no priority algorithm can achieve a constant approximation. (We have already noted how partial enumeration greedy can achieve a PTAS.)
3. For the set cover problem, the natural greedy algorithm is essentially the best priority algorithm.
4. As previously mentioned, for fixed order priority algorithms, there is an $\Omega(\log m / \log \log m)$ inapproximation bound for the makespan problem in the restricted machines model.

Greedy algorithms for the set packing problem

The set packing problem

We are given n subsets $S_1, \ldots, S_n$ from a universe U of size m. In the weighted case, each subset S_i has a weight w_i. The goal is to choose a disjoint subcollection $\mathcal{S}$ of the subsets so as to maximize $\sum_{S_i \in \mathcal{S}} w_i$. In the s-set packing problem we have $|S_i| \leq s$ for all i.

- This is a well studied problem and by reduction from the max clique problem, there is an $m^{\frac{1}{2}-\epsilon}$ hardness of approximation assuming $NP \neq ZPP$. For s-set packing, there is an $\Omega(s/\log s)$ hardness of approximation assuming $P \neq NP$.
- Set packing is the underlying allocation problem in what are called combinatorial auctions as studied in mechanism design.
- We will consider two "natural" greedy algorithms for the s-set packing problem and a somewhat less obvious greedy algorithm for the set packing problem. These greedy algorithms are all fixed order priority algorithms.

The first natural greedy algorithm for set packing

Greedy-by-weight (*Greedy*$_{wt}$

Sort the sets so that $w_1 \geq w_2 \ldots \geq w_n$.
$\mathcal{S} := \emptyset$
For $i : 1 \ldots n$
    If S_I does not intersect any set in $\mathcal{S}$ then
        $\mathcal{S} := \mathcal{S} \cup S_i$.
End For

- In the unweighted case (i.e. $\forall i, w_i = 1$), this is an online algorithm.
- In the weighted (and hence also unweighted) case, greedy-by-weight provides an s-approximation for the s-set packing problem.
- The approximation bound can be shown by a charging argument where the weight of every set in an optimal solution is charged to the first set in the greedy solution with which it intersects.

The second natural greedy algorithm for set packing

Greedy-by-weight-per-size

Sort the sets so that $w_1/|S_1| \geq w_2/|S_2| \ldots \geq w_n/|S_n|$.
$\mathcal{S} := \varnothing$
For $i : 1 \ldots n$
    If S_I does not intersect any set in $\mathcal{S}$ then
        $\mathcal{S} := \mathcal{S} \cup S_i$.
End For

- In the weighted case, greedy-by-weight provides an s-approximation for the s-set packing problem.
- For both greedy algorithms, the approximation ratio is tight; that is, there are examples where this is essentially the approximation. In particular, greedy-by-weight-per-size is only an m-approximation where $m = |U|$.
- We usually assume $n >> m$ and note that by just selecting the set of largest weight, we obtain an n-approximation.