Agenda:

- Greedy Algorithms
  - Fractional Knapsack
  - Job Scheduling on Multi-processors
  - Activity Selection (may be postponed to the following week)

Algorithm Design Techniques:

- Three major design techniques are:
  1. Greedy method
  2. Divide and Conquer
  3. Dynamic Programming

1- Greedy Method:

- This is usually good for optimization problems.
- In an optimization problem we are looking for a solution while we maximize or minimize an objective function. For example, maximizing the profit or minimizing the cost.
- Usually, coming up with a greedy solution is easy; But often it is more difficult to prove that the algorithm is correct.
- General scheme: always make choices that look best at the current step; these choices are final and are not changed later.
- Hope that with every step taken, we are getting closer and closer to an optimal solution.

Example 1: Fractional Knapsack

- We broke into expensive spices store with a knapsack of capacity W.
- We have a set S of n spices to pick, where the store contains a bundle of w_i kilos of spice$_i$ which can be sold in the market for b_i bucks.
- Our goal is to fill the knapsack (without exceeding its capacity) with a combination of the items with maximum profit.
- Example:
  - Knapsack capacity $= 4$kg

	cumin	saffron	pepper	nutmeg	turmeric	paprika
weight (w_i)	2	2	5	1.5	1	3
profit (b_i)	6	8	10	4	3	9

  - One solution: 4kg of pepper (profit $=$ \$8)
  - Optimal solutions: $\{2\text{kg cumin}, 2\text{kg saffron}\}$ or $\{2\text{kg saffron}, 0.5\text{kg cumin}, 1\text{kg turmeric}, 0.5\text{kg paprika}\}$ (profit $=$ \$14)
- Formally, find $0 \leq x_i \leq w_i$ for $1 \leq i \leq n$ such that $\sum_{i=1}^{n} x_i \leq W$ and $\sum_{i=1}^{n} \frac{x_i}{w_i} \times b_i$ is maximized.

Fractional Knapsack: General Description

- Suppose we have a set S of n items, each with a profit/value b_i and weight w_i.
- We also have a knapsack of capacity W,
- Assume that each item can be picked at any fraction, that is we can pick $0 \leq x_i \leq w_i$ amount of item i.
- Our goal is to fill the knapsack (without exceeding its capacity) with a combination of the items with maximum profit.
- Formally, find $0 \leq x_i \leq w_i$ for $1 \leq i \leq n$ such that $\sum_{i=1}^{n} x_i \leq W$ and $\sum_{i=1}^{n} \frac{x_i}{w_i} \times b_i$ is maximized.
- Greedy idea: start picking the items with more "value":

$$\text{value} \equiv \frac{b_i}{w_i}$$

So let $v_i = \frac{b_i}{w_i}$. The algorithm will be as follows:

- The pseudocode is:

```
Procedure Frac-Knapsack (S, W)

for i ← 1 to n do
    x_i ← 0
    v_i ← b_i / w_i
CurrentW ← 0
While CurrentW < W do
    let i be the next item in S with highest value
    x_i ← min{w_i, W − CurrentW}
    add x_i amount of i to knapsack
    CurrentW ← CurrentW + x_i
```

- How to find next highest value in each step?
- One way is to sort S at the begining based on v_i's in non-increasing order.
- Another way is to keep a PQ (max-heap) based on values.
- Since we check at most n items, running time is upper bounded by $O(n \log n)$.
- Since in the WC we check all n items, the lower bound is $\Omega(n \log n)$.
- Hence, running time is $\Theta(n \log n)$.

When and Why Does the Greedy Method Work?

- First, we have the optimal substructure property: the optimal solution contains optimal solutions to sub-problems that look just like the original problem. (We are going to see more of this in dynamic programming later.)
  - Once we decide on x_j then we need to solve *the same problem* on the remaining instance:

$$OPT\Big(W, \langle w_1, b_1\rangle, ..., \langle w_n, b_n\rangle)\Big) =$$
$$\max_j \left\{ \frac{b_j}{w_j} x_j + OPT\Big(W - x_j,\ \langle w_1, b_1\rangle, ..., \langle w_{j-1}, b_{j-1}\rangle, \langle w_{j+1}, b_{j+1}\rangle, ..., \langle w_n, b_n\rangle\Big)\right\}$$

  - It says: "the optimal solution overall must be of the form: taking x_j from some commodity, and filling the remaining $W - x_j$ room in the knapsack with the optimal set of the remaining $n - 1$ spices."
- This already points to a recursive nature of the solution (see Dynamic Programming later for a full understanding):

`RecursiveOutline`(W, S)

1. Let C be a n-integer array
2. For every j:
   - 2.1 Take as much of item j as you can: $x_j \leftarrow \min\{W, w_j\}$
   - 2.2 $C[j] \leftarrow x_j \cdot \frac{b_j}{w_j} +$ `RecursiveOutline`$(W - x_j, S \setminus \{j\})$
3. Pick $\max_j C[j]$

When and Why Does the Greedy Method Work?

- Second, the substitution property for the problem holds:
  Any optimal solution to the problem can be altered to include our greedy choices without hindering optimality.
- It is also called *greedy-choice property* (CLRS p.424; do not worry about relations with dynamic programming (DP); come back to read p.424 after we introduce DP.)
- Here is what it means.
  - Let $S = \{y_1, \ldots, y_n\}$ be an optimal solution and $S' = \{x_1, \ldots, x_n\}$ a solution generated by our algorithm.
  - We have $value(S) \geq value(S')$.
    Note: S is optimal (assumed) and we want to prove so is S'.
  - Below, the cost of a solution S, cost(S), means the same as profit of a solution, the same as value of a solution, $value(S)$. If $S = \{x_1, \ldots, x_n\}$, this is the value $\sum_j x_j \cdot \frac{b_j}{w_j}$.
  - We want to show: each greedy choice in S' can be used to replace some portions of one or more items in S so the altered solution of S, denoted $Z = \{z_1, \ldots, z_n\}$, is still optimal. We will deal with one such item at a time.

When and Why Does the Greedy Method Work?

- In our greedy algorithm's solution S', $x_1 = \min\{W, w_1\}$, meaning item 1 is *saturated* — we cannot pick anymore of item 1.
- Claim: There exists an optimal solution to the Fractional Knapsack problem where item 1 is saturated.
- Proof: Assume the problem's input is $(W, \langle b_1, w_1\rangle, ..., \langle b_n, w_n\rangle)$ and $S = (y_1, ..., y_n)$ is an optimal solution. For simplicity, assume items are sorted by value: $\frac{b_1}{w_1} \geq \frac{b_2}{w_2} \geq ... \geq \frac{b_n}{w_n}$
  - If $y_1 = x_1$ (item 1 is saturated) we are done.
  - So assume item 1 isn't saturated.
  - We alter S to obtain a solution $Z = (z_1, ..., z_n)$ with (i) cost$(Z) \geq$ cost$(S) = OPT$ and with (ii) item 1 saturated (i.e. $z_1 = \min\{W, w_1\}$).
    - Case 1: There's still room in the knapsack, namely $\sum_i y_i < W$.
    - We simply add more of item 1 as much as we can.
    - Set $\Delta = \min\{W - \sum_i y_i,\ x_1 - y_1\}$ (how much room is left in the knapsack and how much more of item 1 we can take).
    - Set $z_1 = y_1 + \Delta$ and $z_j = y_j$ for any other j.
    - By taking more of item 1 the cost of the new solution $(z_1, ..., z_n)$ cannot be smaller than the cost of the $(y_1, .., y_n) = OPT$:

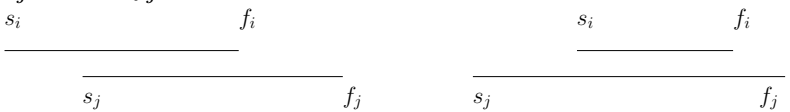
$$\sum_j z_j \cdot \frac{b_j}{w_j} - \sum_j y_j \cdot \frac{b_j}{w_j} = \Delta \cdot \frac{b_1}{w_1} \geq 0$$

When and Why Does the Greedy Method Work (Cont'd)

- Claim: There exists an optimal solution to the Fractional Knapsack problem where item 1 is saturated.
- Proof (Cont'd):
  - Case 2: The knapsack is full, namely $\sum_i y_i = W$.
    Item 1 isn't saturated, so $y_1 < W$, so there must be some $y_k > 0$. We alter the solution to take less of item k and more of item 1 as much as we can:
    - Set $\Delta = \min\{y_k, (x_1 - y_1)\}$ (how much of item k we can discard, and how much more item 1 we can add).
    - Set $z_1 = y_1 + \Delta$, $z_k = y_k - \Delta$ and $z_j = y_j$ for any other item j.
  - This cannot decrease the cost $\sum_j z_j \cdot \frac{b_j}{w_j} - \sum_j y_j \cdot \frac{b_j}{w_j} = \Delta \cdot (\frac{b_1}{w_1} - \frac{b_k}{w_k}) \geq 0$
  - If item 1 is now saturated, we are done. Otherwise, we continue with Case 2 until it become saturated.
- We can repeat the same argument to show that S can be altered using each of the greedy choices in S' so that the resulting solution is optimal.

Example 2: Scheduling on Multiple Processors

- Suppose we have a set T of n jobs, each job i has a start time s_i and finish time f_i
- Each job/task can be performed on one machine and each machine can perform one job at a time
- We say two jobs i and j conflict if their times overlap:
  - Either $s_i \leq s_j < f_i$ or
  - $s_j \leq s_i < f_j$



- Goal: We would like to use minimum number of processors to schedule these jobs on such that all the jobs can run and no two jobs on a machine conflict.
- As an example, think of a conference with different presentations at different start/finish times that need to run. What is the minimum number of rooms we need to have all these presentations in?

- Idea: Schedule a job on one of the previously used machines, take a new machine only if the new job conflicts with some job from each machine already being used.
- Question: In what order do we consider the jobs?
- Suppose we look at the jobs in non-decreasing order of their start time, so say

$$s_1 \leq s_2 \leq s_3 \ldots \leq s_n$$

- `procedure Multi-Job-Schedule` (T)

```
m ← 1
While T ≠ ∅ Do
   extract next job i with smallest s_i value from T
   If i has no conflict with the last job on one of the m machines in use
      Schedule i at the end of that machine
   else
      m ← m + 1
      schedule i on the new machine m
```

More details of steps

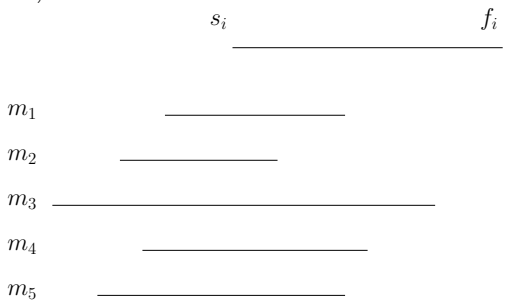
- We can sort the jobs at the begining of the algrithm based on s_i in time $O(n \log n)$
- Then in each iteration of the while loop we can simply take the next job.
- Alternatively we can build a Min-heap on jobs based on s_i and in each iteration do extract-min; This too takes n iterations of $O(\log n)$ cost for each extract-min
- Either way, the cost of finding minimum s_i's one by one is $O(n \log n)$
- What is the time of finding if there is a conflict for each job?
- We should check the finish time of the last job on each machine with the start time of the new job
- If the smallest finish time of jobs on all the machines is $\leq s_i$ then we can schedule job i; else we need a new machine.
- How do we find this out?

- We should maintain what is the finish time of last job on each machine so far.
- We make another PQ (min-heap) for the finish time of last jobs on the machines.
- Each time a new job i is added to a machine j we should increase the finish time of the last job on machine j to finish time of i which is f_i.
- So each step takes $O(\log n)$ to find/update the finish times. There are n iterations. So total time will be $O(n \log n)$.

Correctness of the algorithm:

- Clearly all the jobs on each machine are non-confcliting and we schedule all the jobs (why? use Loop Invariant)
- Need to prove that at the end we use minimum number of machines as well.

- By way of contradiction suppose optimum solution uses k machines while our greedy algorithm uses $k' > k$.
- Consider the first time we put a job i on machine $k+1$.
- So at this time, job i conflicts with all the "last" jobs on machines $1, \ldots, k$.



- So all these (last) jobs start before s_i and finish after s_i.
- So at time s_i, all these k jobs plus job i must run in parallel
- Therefore, we need at least $k+1$ processors.
- Thus optimum cannot use only k processors to run even just these $k+1$ jobs.

Example 3: Activity Selection

- Suppose that we have n jobs, each with a start time s_i and finish time f_i; $0 \leq s_i < f_i$; these are given and are fixed.
- We have only one machine to use and the machine can process one job at a time (and once started, a job most be run until its finish time).
- Two jobs i and j are said to conflict (or are incompatible) if the two intervals $[s_i, f_i)$ and $[s_j, f_j)$ overlap.
- Goal: schedule a largest subset of non-conflicting jobs on this machine
- 1st attempt: suppose we sort the jobs based on start time, i.e. $s_1 \leq s_2 \leq \ldots \leq s_n$. Schedule a job if we can.
  - Bad example: $s_1 = 1$, $s_2 = 2$, $s_3 = 3$
    $f_1 = 4$, $f_2 = 3$, $f_3 = 4$
    The greedy can only schedule job 1 but the optimal is to schedule 2 and 3.
- 2nd attempt: Try to schedule shorter jobs first, i.e. sort jobs based on $f_i - s_i$ in non-decreasing order and then Schedule a job if we can.
  - Bad example: $s_1 = 3$, $s_2 = 1$, $s_4 = 4$
    $f_1 = 5$, $f_2 = 4$, $f_3 = 8$
    The greedy can only schedule job 1 but the optimal is to schedule 2 and 3.
- 3rd attempt: Sort the jobs based in non-decreasing order of finish time, i.e. $f_1 \leq f_2 \leq \ldots \leq f_n$ and then Schedule a job if we can.

There exists at least one optimal sol that starts with the greedy choice

Correctness:

```
procedure Activity-Selection (S)

   Sort activities in S s.t.  f_1 ≤ f_2 ≤ ... ≤ f_n
   A ← ∅; e ← 0;
   for i ← 1 to n do
      if s_i ≥ e then
         A ← A ∪ {i}
         e ← f_i
   return A
```

- Note that the running time of this algorithm is $\Theta(n \log n)$ as it's the time needed to sort the jobs and each iteration of the loop takes constant time.
- In this algorithm A is the set of jobs scheduled by our algorithm and e at any given time is the time at which the last scheduled job finishes (i.e. the earliest time we can schedule the next job).
- We prove that this 3rd attempt is correct.
- First note that, since we always schedule a new job if its start time s_i is larger than the finish time of the last scheduled job we get a set of compatible jobs at the end.

- To complete correctness, we have to show our algorithm obtains an optimum solution.
- Promising: we say a schedule after step i is promising if it can be extended to an optimal schedule using a subset of jobs in $\{i+1, \ldots, n\}$.
- We prove that after every step i, the partial solution we have is promising.
- Formally, let A_i and e_i denote the values of A and e after iteration i of the for-loop, respectively.
- Note that $A_0 = \emptyset$ and $e = 0$.
- We show that after each iteration $i \geq 0$, the decisions we have made so far are all correct in the sense that there is an optimum solution which contains everything we have selected so far and everything that we have decided not to include (so far) does not belong to that optimum either: **Lemma:** For every $0 \leq i \leq n$, there is an optimal solution A_{opt} such that $A_i \subseteq A_{opt} \subseteq A_i \cup \{i+1, \ldots, n\}$.
- Note that if we prove this for all $0 \leq i \leq n$, and in particular for $i = n$ we have: $A_n \subseteq A_{opt} \subseteq A_n \cup \emptyset$ which implies $A_n = A_{opt}$, i.e. our solution is the same as some optimum solution; this is what we wanted.
- We use induction on i to prove the above lemma.
- Base: $i = 0$, clearly empty schedule can be extended to an optimal one from $\{1, \ldots, n\}$; Thus $\emptyset = A_0 \subseteq A_{opt} \subseteq \emptyset \cup \{1, \ldots, n\}$.

- Induction Step: Suppose we have a promising schedule after step $i \geq$, i.e. there is an optimum solution A_{opt} such that:
  $A_i \subseteq A_{opt} \subseteq A_i \cup \{i+1, \ldots, n\}$.
- We consider three cases
- **Case 1:** if $s_{i+1} < e_i$ so we cannot schedule job $i+1$ because it overlaps with one of the previously scheduled ones; i.e. $A_{i+1} = A_i$.
  Since $A_i \subseteq A_{opt}$, A_{opt} has the job from A_i that is overlapping with $i+1$ as well, so A_{opt} cannot have $i+1$ either. Thus:
  $A_{i+1} = A_i \subseteq A_{opt} \subseteq A_{i+1} \cup \{i+2, \ldots, n\}$.
- **Case 2:** $s_{i+1} \geq e_i$; so $A_{i+1} = A_i \cup \{i+1\}$.
  Case 2A: $i+1 \in A_{opt}$ then
  $A_{i+1} \subseteq A_{opt} \subseteq A_{i+1} \cup \{i+2, \ldots, n\}$ and we are done.
  Case 2B: $i+1 \notin A_{opt}$;
  - Note that $s_{i+1} \geq e_i$ and so $i+1$ does not conflict with anything in A_i;
  - On the other hand we cannot add $i+1$ to A_{opt} (otherwise it would have not been an optimum solution); so there must be a job $j \in A_{opt}$ conflicting with $i+1$. Since $A_{opt} \subseteq A_i \cup \{i+1, \ldots, n\}$ and since $i+1$ does not conflict with anything in A_i: $j \geq i+2$.
  - Let $j \geq i+2$ be a job of A_{opt} with the earliest finish time that is conflicting with $i+1$.
  - All activities in $A_{opt} - A_i - \{j\}$ have start-time after f_j (or they will overlap with job j); so they do not overlap with job $i+1$ because $i+1$ finishes before f_j.
  - So $A' = (A_{opt} - \{j\}) \cup \{i+1\}$ is feasible and has the

w_i	5	1	4	2
S_1:	2	1	4	2
$S_{1,1}$:	5	1	3	0

w_i	5	3	4	2
S_1	3	3	4	0
S_2	5	3	2	0
S_3	5	1	4	0

let $S = \{y_1, \ldots, y_n\}$ be an opt. solution

$S' = \{x_1, \ldots, x_n\}$ our solution