1.204 Lecture 10

Greedy algorithms:
Knapsack (capital budgeting)
Job scheduling

Greedy method

- Local improvement method
  - Does not look at problem globally
  - Takes best immediate step to find a solution
  - Useful in many cases where
    - Objectives or constraints are uncertain, or
    - An approximate answer is all that's required
  - Generally O(n) complexity, easy to implement and interpret results
    - Often requires sorting the data first, which is O(n lg n)
  - In some cases, greedy algorithms provide optimal solutions (shortest paths, spanning trees, some job scheduling problems)
    - In most cases they are approximate algorithms
    - Sometimes used as a part of an exact algorithm (e.g., as a relaxation in an integer programming algorithm)

General greedy algorithm

```
// Pseudocode
public solution greedy(problem) {
   solution= empty set;
   problem.sort();     // Usually place elements in order
   for (element: problem) {
        if (element feasible and appears optimal)
          solution= union(solution, element);
   return solution;
}

Some greedy algorithms sort, some use a heap, some don't need
   to sort at all.
```

Greedy knapsack problem

We have n objects, each with weight w_i and profit p_i. The knapsack has capacity M.

$$\max \sum_{0 \le i < n} p_i x_i$$

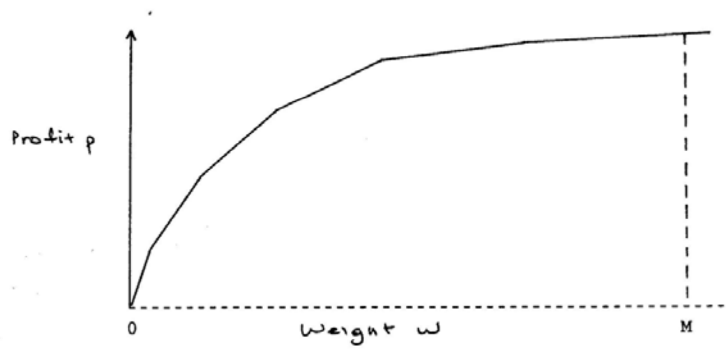
$$s.t.$$

$$\sum_{0 \le i < n} w_i x_i \le M$$

$$0 \le x_i <= 1$$

$$p_i \ge 0, w_i \ge 0, \; 0 \le i < n$$

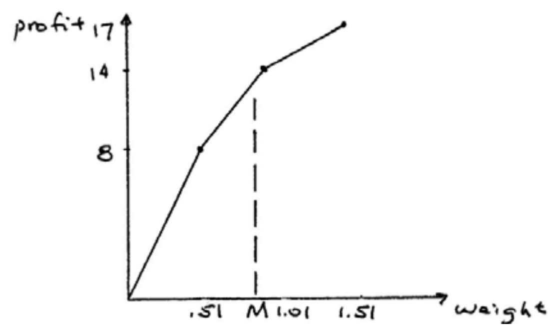
Greedy knapsack algorithm



Algorithm chooses element with highest value/weight ratio first, the next highest second, and so on until it reaches the capacity of the knapsack.
This is the same as a gradient or derivative method.

Knapsack: integer or not?

element	weight	profit
1	.51	8
2	.5	6
3	.5	3



Let M= 1.
Integer solution is {2, 3}, an unexpected result in some contexts.
Greedy solution is {1, 98% of 2}.
If problem has hard constraints, need integer solution.
If constraints are fuzzy, greedy solution may be better.

Knapsack problems

- Truck packing: integer knapsack
  - Packing problem in 2 and 3 dimensions is extension
- Investment program:
  - Greedy knapsack at high level
  - Can be integer knapsack at individual transaction level
  - (Highway investment or telecom capital investment programs often handled as integer problem, with occasionally hard-to-interpret results)
  - Used to train telecom execs for spectrum auction
- Interactions between projects:
  - Greedy can be extended to handle interactions between small numbers of projects (that can be enumerated)
  - Integer program handles this explicitly

Greedy knapsack code, p.1

```
public class Knapsack {
   private static class Item implements Comparable {
        public double ratio;          // Profit/weight ratio
        public int weight;
        public Item(double r, int w) {
                ratio = r;
                weight = w;
        }

        public int compareTo(Object o) {
                Item other = (Item) o;
                if (ratio > other.ratio)     // Descending sort
                        return -1;
                else if (ratio < other.ratio)
                        return 1;
                else
                        return 0;
        }
   }
```

Greedy knapsack code, p.2

```
public static double[] knapsack(Item[] e, int m) {
     int upper = m;         // Knapsack capacity
     // 0-1 answer array: 1 if item in knapsack, 0 if not
     double[] x= new double[e.length];
     int i;
     for (i= 0; i < e.length; i++) {
         if (e[i].weight > upper)
            break;
         x[i]= 1.0;
         upper -= e[i].weight;
     }
     if (i < e.length)      // If all items not in knapsack
         x[i]= (double) upper/ e[i].weight; // Fractional item
     return x;
}
```

Greedy knapsack code, p.3

```
public static void main(String[] args) {
     Item a = new Item(2.0, 2);
     Item b = new Item(1.5, 4);
     Item c = new Item(2.5, 2);
     Item d = new Item(1.66667, 3);
     Item[] e = { a, b, c, d };
     Arrays.sort(e);
     int m = 7;
     System.out.println("Capacity: " + m);
     double[] projectSet= knapsack(e, m);
     double cumProfit= 0.0;
     for (int i= 0; i < e.length; i++) {
            System.out.println( … );      // See Java code
            cumProfit+= projectSet[i]*e[i].weight*e[i].ratio;
     }
     System.out.println("Cumulative benefit: " + cumProfit);
}
```

Greedy knapsack output

```
Capacity: 7

i:  ratio: 2.5  wgt: 2  profit: 5.0    in? 1.0
i:  ratio: 2.0  wgt: 2  profit: 4.0    in? 1.0
i:  ratio: 1.67 wgt: 3  profit: 5.0    in? 1.0
i:  ratio: 1.5  wgt: 4  profit: 6.0    in? 0.0

Cumulative benefit: 14.0

(Roundoff errors omitted)

This greedy example yields an integer solution. Most don't:
Run knapsack() with m= 6 or 8 or …
```

Greedy job scheduling

- **We have a set of n jobs to run on a processor (CPU) or machine**
- **Each job i has a deadline d_i >=1 and profit p_i >=0**
- **There is one processor or machine**
- **Each job takes 1 unit of time (simplification)**
- **We earn the profit if and only if the job is completed by its deadline**
  - **"Profit" can be the priority of the task in a real time system that discards tasks that cannot be completed by their deadline**
- **We want to find the subset of jobs that maximizes our profit**

- **This is a restricted version of a general job scheduling problem, which is an integer programming problem**
  - **Example use in telecom engineering and construction scheduling**
  - **Many small jobs, "profit" proportional to customers served**
  - **This is then combined with integer programming solution for big jobs**
- **Greedy also used in how many machines/people problems (hw 1)**
  - **Buy versus contract**

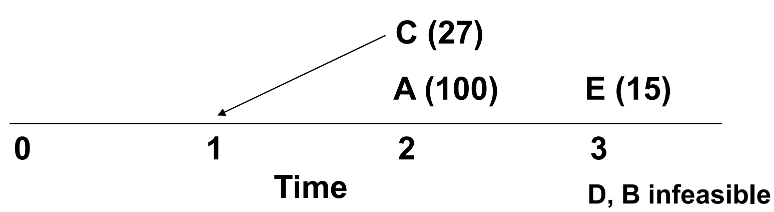
Greedy job scheduling example

Number of jobs n=5. Time slots 1, 2, 3. (Slot 0 is sentinel)

Job (i)	Profit	Deadline	Profit/Time
A	100	2	100
B	19	1	19
C	27	2	27
D	25	1	25
E	15	3	15

Greedy job scheduling algorithm

- **Sort jobs by profit/time ratio (slope or derivative):**
  - A (deadline 2), C (2), D (1), B (1), E (3)
- **Place each job at latest time that meets its deadline**
  - Nothing is gained by scheduling it earlier, and scheduling it earlier could prevent another more profitable job from being done
  - Solution is {C, A, E} with profit of 142



  - This can be subproblem: how many machines/people needed

Greedy job data structure

- Simple greedy job algorithm spends much time looking for latest slot a job can use, especially as algorithm progresses and many slots are filled.
  - n jobs would, on average, search n/2 slots
  - This would be an $O(n^2)$ algorithm
- By using our set data structure, it becomes nearly O(n)
  - Recall set find and union are O(Ackermann's function), which is nearly O(1)
  - We invoke n set finds and unions in our greedy algorithm

Simple job scheduling: $O(n^2)$

```
public static int[] simpleJobSched(Item[] jobs) {
     int n= jobs.length;
     int[] jobSet= new int[n];
     boolean[] slot= new boolean[n];
     for (int i= 1; i < n; i++) {
            for (int j= jobs[i].deadline; j > 0; j--) {
                   if (!slot[j]) {
                          slot[j]= true;
                          jobSet[j]= i;
                          break;
                   }
            }
     }
     return jobSet;
}
```

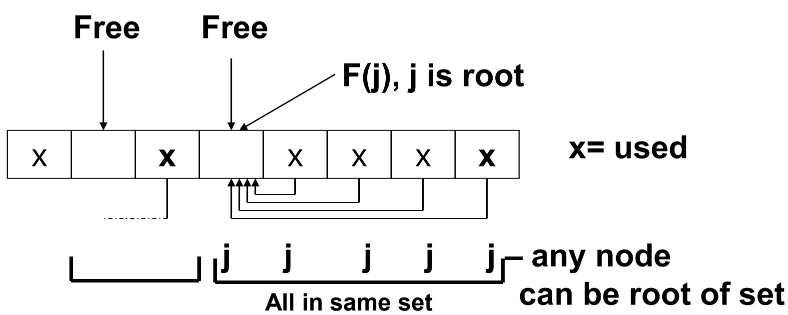
Fast job scheduling (almost O(n))

- **We use i to denote time slot i**
  - **At the start of the method, each time slot i is its own <u>set</u>**
- **There are b time slots, where b= min{n, max(d_i)}**
  - **Usually b= max(d_i), the latest deadline**
- **Each set k of slots has a value F(k) for all slots i in set k**
  - **This stores the highest free slot before this time**
  - **F(k) is defined only for root nodes in sets**

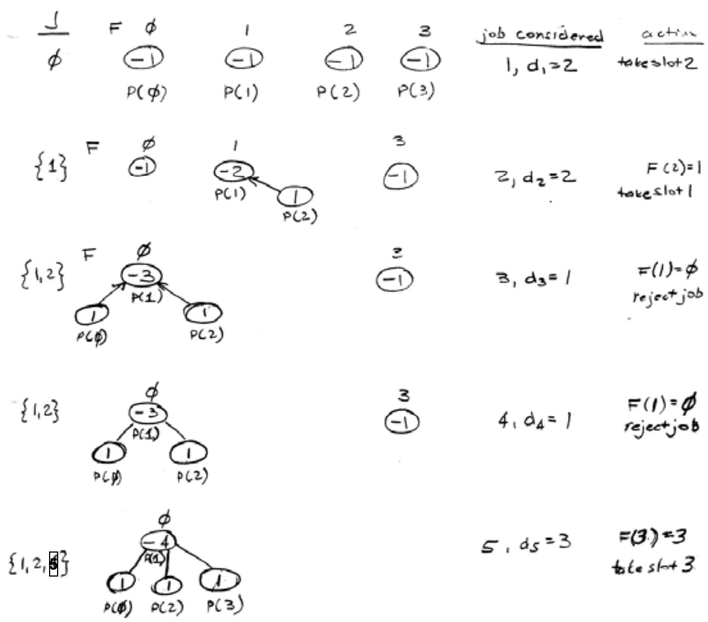
Job scheduling algorithm

- **Initially all slots are free**
  - **We have b+1 sets corresponding to b+1 time slots i, $0 \le i \le b$**
  - **Slot 0 is a sentinel**
  - **Initially F(i)= i for all i**
- **We will use parent p[i] to link slot i into its set**
  - **Initially p[i]= -1 for all i**
  - **Parent of root is negative of number of nodes in set**
- **To schedule job i with deadline d_i:**
  - **"Find" root of tree containing slot min(n, d_i)**
    - **Usually this is just slot d_i**
  - **If root of i's set is j, then F(j) is latest free slot, provided F(j) ≠ 0**
- **After using slot F(j), we combine ("set union") set having root j with set having slot F(j) -1**

Job scheduling example



Job scheduling algorithm operation



Job sequence code, p.1

```
public class JobSeqFast {
   private static class Item implements Comparable {
        private int profit;
        private int deadline;
        private String name;
        public Item(int p, int d, String n) {
                profit= p;
                deadline= d;
                name= n;
        }
        public int compareTo(Object o) {
                Item other = (Item) o;
                if (profit > other.profit)    // Descending sort
                        return -1;
                else if (profit < other.profit)
                        return 1;
                else
                        return 0;
        }
   }    // Add getXXX() and setXXX() methods for completeness
```

Job sequence code, p.2

```
public static int[] fjs(Item[] jobs, int b) {
  int n= jobs.length;
  int[] j= new int[n];     // Profit max jobs, in time order
  Set jobSet = new Set(b);
  int[] f = new int[b];    // Highest free slot, job due at i
  for (int i = 0; i < b; i++)
    f[i] = i;              // Sentinel at jobs[0]

  for (int i = 1; i < n; i++) {    // Jobs in profit order
    int q = jobSet.collapsingFind(Math.min(n, jobs[i].deadline));
    if (f[q] != 0) {              // If free slot exists
      j[q] = i;                   // Add job in that slot
      int m = jobSet.collapsingFind(f[q] - 1);  // Find earlier slot
      jobSet.weightedUnion(m, q); // Unite sets
      f[q] = f[m];                // In case q is root, not m
    }
  }
  return j;                       // Jobs in optimal set
}                                 // More comments in download code
```

Job sequence code, p.3

```
public static void main(String args[]) {
     Item sentinel= new Item(0, 0,"s");// Don't sort-leave in place
     Item a = new Item(100, 2, "a");  // Also create b, c, d, e
     Item[] jobs = { sentinel, a, b, c, d, e };
     Arrays.sort(jobs, 1, jobs.length-1);  // Sort descending
     int maxD= -1;                 // Maximum deadline
     for (Item i: jobs)
       if (i.deadline > maxD)
         maxD= i.deadline;
     maxD++;
     int bb= Math.min(maxD, jobs.length);
     int[] j= fjs(jobs, bb);
     System.out.println("Jobs done:");
     for (int i= 1; i < maxD; i++) {
       if (j[i]>0) {
         System.out.println(" Job "+ jobs[j[i]].name +
              " at time "+ i);
}    // And compute and output total jobs, total profit
```

Job sequence example output

```
Jobs done:
 Job c at time 1
 Job a at time 2
 Job e at time 3
Number of jobs done: 3, total profit: 142
```

Summary

- This job scheduling special case solvable with greedy algorithm
  - We revisit more general version with dynamic programming
- Capital planning problems often solvable with greedy algorithm
- Other greedy algorithms
  - Spanning trees (next time)
  - Shortest paths (in two lectures)
  - Other job scheduling problems (e.g. min time schedule)
  - Graph coloring heuristic
  - Traveling salesperson heuristic (2-opt, 3-opt)
    - Used as part of simulated annealing
- Greedy algorithms are fast and relatively simple
  - Consider them as parts of more complex solutions, or
  - As approximate solutions