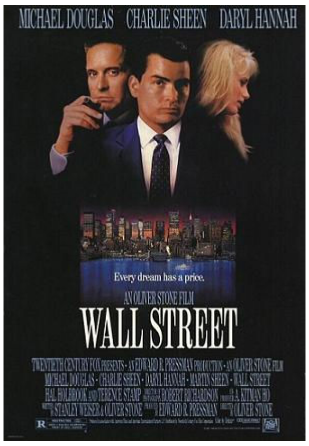
Greedy Algorithms

“Greed is good.
Greed is right.
Greed works.
Greed cuts through,
clarifies, and
captures the
essence of the
evolutionary spirit.”

- Gordon Gekko

(Michael Douglas)



http://en.wikipedia.org/wiki/Wall_Street_(film)

Greedy Algorithms

- Simple idea, but it doesn’t always work.
  - Used to solve optimization problems that involve choosing objects, events, (some type of entities) from a set.
- The Greedy Strategy:
  - At each step, make the “best” next choice.
  - Never backtrack or change past choices.
  - Never look ahead to see if this choice has negative consequences.
- Depending on the problem: this may or may not give the correct answer.
- The text treats greedy algorithms as a special case of dynamic programming
  - The greedy paradigm is simpler
  - We will do greedy algorithms first

Properties of Greedy Algorithms

- Usually simple and fast (often linear time).
- Hard part is demonstrating optimality.
- Examples: Huffman encoding, some sorting algorithms.
- Typically applied to optimization problems (find solution which maximizes or minimizes some objective function).

Example: Making Change

- Input:
  - Suppose we are given the specification of a monetary denominational system
    - (a list of the values of different coins that are in use, for example: 1, 5, 10, 25, 100, 200 cents).
  - We are also given an amount (specified in cents) to return to a customer
- Problem:
  - Do this using the fewest number of coins

Alexandrian Coin Denominations

- Used in AD 117-138
  - Time of emperor Hadrian

The Coin Changing Problem

- Assume that we have an unlimited number of coins of various denominations:
  - 1c (pennies), 5c (nickels), 10c (dimes), 25c (quarters), 1$ (loonies)
- Objective: Pay out a given sum S with the smallest number of coins possible.

- The greedy coin changing algorithm:
  - This is a $\Theta(m)$ algorithm where m = number of denominations.

```
while S > 0 do
   c := value of the largest coin no larger than S;
   num := S div c;
   pay out num coins of value c;
   S := S - num*c;
```

Example: Making Change

- E.g.:

  $5.64 = $2 +$2 + $1 +
  .25 + .25 + .10 +
  .01 + .01 + .01 +.01

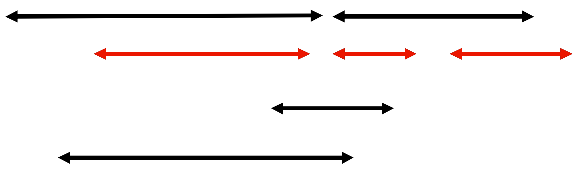
- Nontrivial exercise: prove this always uses smallest number of coins
- Easier exercise:
  set up a monetary system for which this algorithm fails

Framework for Greedy Algorithms

- At each stage, we have a partial solution which can be extended to a larger solution.
  - We have a **set of choices** for this next step and we select a “good” choice to extend the solution.
- We have some sort of predefined measure of what a good choice is.
  - Take the “best” choice under that measure.
- Repeat until a complete solution is obtained.

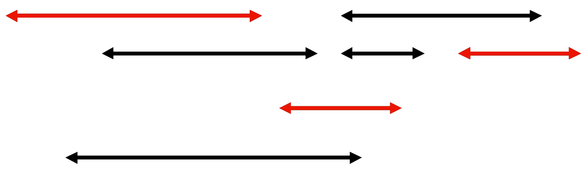
Scheduling activities

- Instance: a set of n activities, each with start time s_i and finishing time f_i
- Problem: find the subset of activities with the maximum number of non-overlapping activities.



Scheduling Activities (cont.)

- We say activities i and j are *compatible* if they do **not** overlap (i.e.: $s_i \geq f_j$ or $s_j \geq f_i$)
- Idea: choose the activity with earliest finishing time, delete it and those that are incompatible with the chosen activity, then repeat…



A Scheduling Algorithm

- Sort activities by finish time.
- Choose first activity.
- Repeatedly choose the next activity that is compatible with all previously chosen ones.

- Running time: $\Theta(n \log n)$ time to sort, $\Theta(n)$ time for the rest.
- How do we prove this is correct?

Notation

- Let $S = \{a_1, a_2, \ldots, a_n\}$ be the list of activities.
- Assume each activity is represented by a half-open interval, i.e.: $a_i = [s(a_i), f(a_i))$.
- Sort the list of activities such that

  $f(a_1) \leq f(a_2) \leq f(a_3) \leq \ldots \leq f(a_n)$.
- So compatibility of a_i and a_k implies that: $s(a_k) \geq f(a_i)$ or $s(a_i) \geq f(a_k)$.
  - That is, they do not overlap.

The Greedy Algorithm Objective

- Greedy selection scans the sorted activities and always chooses the **earliest** finishing activity that is compatible with all previously selected activities.
  - The scan has cost O(n).
  - At each step this greedy strategy will **maximize** the amount of unscheduled remaining time.
- We now show that this strategy will derive an optimal set of activities (it is impossible to find a larger set of activities that are mutually compatible).
  - Note: The activity problem may have several optimal solutions (of course each optimal solution has the same number of activities).

The Greedy Selection Theorem

- Theorem: Greedy selection will produce an optimal solution for the activity selection problem.
- Proof:
  - Suppose greedy selection has chosen activities: $G = \{g_1, g_2, g_3, \ldots, g_m\}$ where each g_i is some a_k.
  - List is sorted by finishing time, i.e. $f(g_i) \le f(g_{i+1})$.
  - We will show by induction that given any optimal solution H, it can be modified (without loss in the number of tasks) to start with $g_1, g_2, g_3, \ldots, g_k$ as the first k selections.
  - So, when $k = n$, the optimal solution H looks exactly like G and has the same number of tasks as H started with. So $|G|=|H|$ and hence G is also optimal.

The Greedy Selection Theorem

- Base case ($k = 1$):
  - Activity g_1 can be the first selection of an optimal solution. To see this consider the following:
  - Suppose we have optimal solution $H = \{h_1, h_2, h_3, \ldots, h_q\}$ where h_1 is not g_1.

Note that $f(g_1)$ would be the first entry in the list sorted by finish times.

  - Then since $f(g_1) \leq f(h_1)$ we could replace h_1 with g_1, maintain compatibility and still have the same number of activities in H (implying that it would still be optimal).
  - Therefore, g_1 can be the start of an optimal solution.

The Greedy Selection Theorem

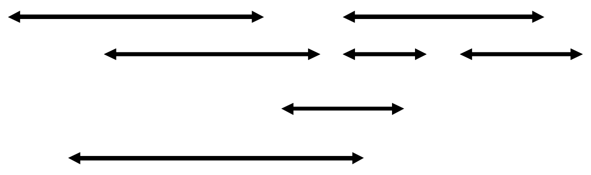
- The inductive step:
  - Assume $g_1, g_2, g_3, \ldots, g_k$ are the first k selections of an optimal solution called $H = \{g_1, g_2, g_3, \ldots, g_k, h_{k+1}, \ldots, h_q\}$.
  - Note that $H^* = \{h_{k+1}, \ldots, h_q\}$ is an optimal solution of the problem with activities: $S^* = \left\{a_i \middle| a_i \in S, s(a_i) \geq f(g_k)\right\}$ because if not then we could find a better solution with more activities for S^* which replace $h_{k+1}, \ldots, h_q$ in H and thus generate a solution with more activities than H !!
  - Now, just as in the base case we can replace h_{k+1} with g_{k+1} and maintain the optimality of H^*.
  - So the next greedy choice g_{k+1} extends the greedy solution and gives an optimal solution starting with $g_1, g_2, \ldots, g_{k+1}$ as required.

The Greedy Selection Theorem

- More details:
  - Why is $H' = \{g_1, g_2, g_3, \ldots, g_k, g_{k+1}, h_{k+2}, \ldots, h_q\}$ optimal?
  - Inspecting this solution we see that all entries are compatible and we note that it has the same number of activities as H. So it is optimal.
- Is it possible for the greedy algorithm to stop before generating a complete optimal solution?
  - No. If we had an optimal solution such as $\{g_1, g_2, g_3, \ldots, g_m, d_{m+1}, \ldots, d_q\}$ with $q > m$ and g_m was the last activity that could be greedily selected by the algorithm, then we would have $s(d_{m+1}) \geq f(g_m)$ which indicates that there is still at least one activity that could be found by the greedy algorithm in contradiction of the fact that it is finished.

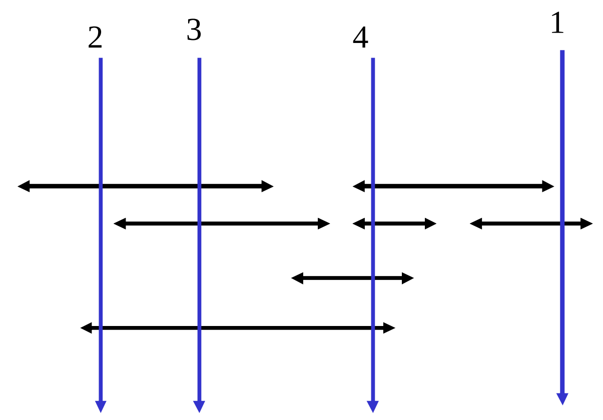
Scheduling Activities into Rooms

- Instance: a set of n classes, each with start time s_i and finishing time f_i
- Problem: find the minimum number of classrooms required to hold all classes



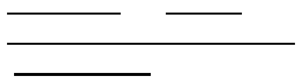
Scheduling Activities into Rooms

- Depth of the graph



A (bad) Room Scheduling Algorithm

- First attempt:
  - Sort activities by finish time (after all we did this for the previous scheduling problem).
  - Start with $i = 1$. Then repeatedly choose next activity and assign to classroom i; next classroom is $i = i + 1 \bmod d$
- How do we prove this is correct?
  - This cannot be done. Beware of bad examples!
  - This “solution” works for the previous slide but in general it fails!



A Room Scheduling Algorithm

- We sort the activities by start time.
  - We still require the depth computation: the minimum number of rooms needed is d. We use d labels to name the rooms.
  - Assuming the activities have been sorted with respect to **start** time we designate the i^{th} activity as a_i.
  - Pseudo-code:

```
For j = 1 to n
  For each $a_i$ that precedes $a_j$ and overlaps it
     Exclude the label of $a_i$ from the set of all labels
  Endfor
  If there is any label in the set of d labels that has
  not been excluded then
   Assign a non-excluded label to $a_j$
  Else Leave $a_j$ unlabeled.
  Endif
Endfor
```

Proof for the Room Scheduling Algorithm (1)

- The following claims prove the success of this greedy algorithm:
- No activity goes unlabeled.
    - The “ELSE statement” will never execute.
  - Consider a_j and assume there are k activities prior to a_j in the sorted order that overlap a_j.
  - These $k+1$ activities (a_j included) form a set that must have a depth $\leq d$ at the start time of a_j.
  - So, $k+1 \leq d$ implies $k \leq d -1$ implies that at the start time of a_j there is a free label (i.e. a free room).

Proof for the Room Scheduling Algorithm (1)

- Also, no two overlapping activities can get the same label.
  - Suppose a_i and a_j overlap.
    - Assume WLOG that a_i precedes a_j in the sorted order.
  - Then when a_j is considered by the algorithm, the label for a_i will be excluded and it cannot be assigned to a_j.
  - Notice how the proof has made use of the three main components of this greedy algorithm: the computation of the depth, the sorting, and the excluding operation.
    - If you had a proof that avoided any of these components then something would be amiss: either the algorithm is suspect or the proof is wrong…

Fractional Knapsack

- A thief breaks into a warehouse full of bulk materials. Each type of material is conveniently labeled with the price per pound.

- The thief has a knapsack of fixed weight capacity and the task is to fill the knapsack in a way that maximizes the value of his loot.

Dollars per pound…

- Reported in Fortune Magazine
  - Mar. 20, 2000, pg. 68:

Product	Price ($)	Weight(lbs)	Price($/lb)
Hot-rolled Steel	370	2000	0.19
Chevrolet sedan	17,700	2630	6.76
Mercedes-Benz 4-door	78,445	4134	18.98
Cigarettes (20)	4	0.04	100.00
Palm V	449	0.26	1,726.92
Hermes Scarf	275	0.14	1,964.29
Gold (ounce)	302	0.0625	4,827.20
Viagra	8	0.00068	11,766.00
Pentium 111 800 MHz	851	0.01984	42,893.00

Fractional Knapsack

- In the knapsack problem we have a knapsack of a fixed capacity (say W pounds) and different items $1,\ldots,n$ each with a given weight w_i and a given benefit b_i.

- We want to put items into the knapsack so as to maximize the benefit subject to the constraint that the sum of the weights must be less than W.

Fractional Knapsack

```
Input:   Set S of $n$ items, weights $w_i$, benefit $b_i > 0$,
         knapsack size $W > 0$.
Output: Amounts $x_i$ maximizing benefit
         with capacity <= W.
for each $i$ in S do
   $x_i \leftarrow 0$
   $v_i \leftarrow b_i/w_i$    // The value "per pound" of item i
$w \leftarrow W$           // Remaining capacity in knapsack
while ($w > 0$) and ($S$ is not empty)
   remove from $S$ item of max value  // Greedy choice
   let $i$ be that item
   $x_i \leftarrow \min(w_i, w)$ // Cannot carry more than $w$ more weight
   $w \leftarrow w - x_i$
```

Fractional Knapsack

Claim. Greedy algorithm gives best optimal item set.

Proof: Sort the items by value per pound.

Let $OPT = \{y_1, y_2, \ldots, y_n\}$ be an optimal solution to fractional knapsack, consisting of amounts y_i for item $i = 1..n$.

We will transform OPT into the greedy solution $\{x_i\}_{1..n}$ using induction while never reducing the value of OPT.

Hence greedy must have the same value as OPT and it is also optimal.

Fractional Knapsack

Proof: WLOG OPT and greedy are sorted by value $\$ / \#$

Basis of induction: $i = 1$. Clearly $y_1 \leq x_1$ since the greedy choice loads as much weight as possible from the most valuable item (price/weight).

If $y_1 = x_1$, then OPT and greedy match in the first selection, and there is nothing to show.

If $y_1 < x_1$ then remove $|x_1 - y_1|$ pounds from the least valuable items starting from y_m and replace them with the same amount of pounds of y_1.
Value of solution cannot go down as y_1 gives the best value per pound.

Fractional Knapsack

Proof:

Step of induction: Assume $y_j = x_j$ for $j < i$.
Observe that $y_i \leq x_i$ since the greedy strategy loads as much weight as possible from item i, given the choices up to j.

If $y_i = x_i$, then OPT and greedy match in their selection, and there is nothing to prove.

If $y_i < x_i$ then remove $|x_i - y_i|$ pounds from the least valuable items starting from y_m and replace them with the same amount of y_i.
Value of solution cannot go down as y_i gives the best value per pound of y_k, $k = i..m$.

Fractional Knapsack

Proof:

Hence the value of OPT and greedy is the same ➔ greedy is optimal too.

Fractional Knapsack

Time analysis:

O($n \log n$)	to sort the items by value
O(n)	to select amount of each item
O($n \log n$)	total

Task Scheduling

Problem: We wish to schedule n tasks on a single CPU. Each task has deadline d_i and requires continuous execution time t_i.

- All deadlines d_i and task times t_i are known at time zero.
- Start times $s(i)$ and finish times $f(i)$ will be determined by the scheduling algorithm.

- Suppose that you are allowed to run late, i.e. a finishing time $f(i) > d_i$ is permissible.
- The lateness of task i is defined as $l_i := f(i) - d_i$.

Task Scheduling

- The maximum lateness of a schedule S is defined as:

$$L(S) \equiv \max_{1 \le i \le n} \{l_i\}.$$

- The goal is to schedule the tasks in such a way as to minimize the maximum lateness

$$\min_S \{L(S)\}$$

among all possible schedules S.

Greedy Task Scheduling

- Shortest jobs first?
  - No! This ignore deadlines. (Counterexample to be given).
- Jobs with smallest slack time $d_i - t_i$?
  - No! (A counterexample can be given).
- Sort by deadline in increasing order (Earliest Deadline First = EDF)
  - Jobs with earlier deadlines finish sooner.
  - Does this work? (We are ignoring the t_i values…).

Greedy Scheduling Algorithm

- Sort task activities by deadline.
  - Let time stamp $Q = 0$
- Repeatedly assign to task $i = 1..n$

  $s(i) = Q$

  $Q = Q + t_i$

  $f(i) = Q$

  - Note that the t_i are used in the Q computations but <u>not</u> in the determination of job order. (!)
- How do we prove this greedy schedule gives an optimal solution?

Correctness of Greedy Algorithm (1)

Claim. The optimal schedule has no idle time.

i.e. No gaps or free time in the schedule.

Proof: Obvious

- Because of the computations for Q one task starts immediately after a different task finishes.

- Our notation: we assume that tasks are labeled by their order in the EDF sort.
  - So: $d_1 \leq d_2 \leq \ldots \leq d_n$

Correctness of Greedy Algorithm (2)

- Definition:
  - If we have a schedule S' such that for some i and j with $d_i < d_j$ we have $task(i)$ scheduled **after** $task(j)$ then we say that S' has an inversion.

Correctness of Greedy Algorithm (3)

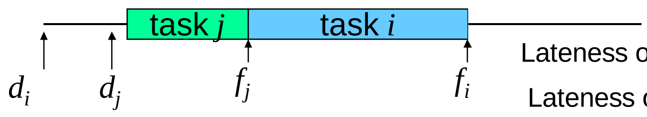
- Claim:
  If schedule S' has an inversion, then there are two tasks: $task(i)$ and $task(j)$ with $d_i < d_j$ but with $task(i)$ scheduled immediately after $task(j)$.
- Proof:
  - Suppose there is at least one inversion.
  - There is $task(b)$ scheduled before $task(a)$ but $d_a < d_b$.
  - If we inspect the schedule going from $task(b)$ towards $task(a)$ we will see a first occurrence when the deadline decreases.
  - This will be our instance of "$task(i)$ scheduled immediately after $task(j)$ and such that $d_i < d_j$".

Correctness of Greedy Algorithm (4)

- Note: all schedules with no inversions and no idle time have the same maximum lateness.
  - With no inversions the schedules can only differ by reordering tasks with the same deadlines.
  - They are all scheduled consecutively.
  - Independent of this schedule the last task in this group determines the maximum lateness and this amount of lateness will be independent of the order within this group.

Correctness of Greedy Algorithm (5)

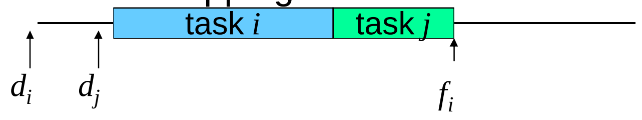
- We use an exchange argument.
  - Suppose there is an optimal solution with an inversion.
  - Then we can find i and j such that: $d_i < d_j$ and $task(i)$ is scheduled immediately after $task(j)$.
  - Swap tasks i and j.
- Claim. Lateness does not increase.
  - Before swapping the optimal solution contains:



Lateness of task j is f_j - d_j

Lateness of task i is $f_i - d_i$

  - After swapping:



Correctness of Greedy Algorithm (6)

- Since task i finishes earlier it cannot be responsible for an increase in lateness in the “after swapping” schedule.
- Does the maximum lateness go up because task j now runs after task i?
- NO: Its finish time is f_i.
  - Its lateness is $f_i - d_j$ and note that:
    $f_i - d_j < f_i - d_i$ because $d_i < d_j$.

New lateness
for Task j.

Lateness for Task i in old schedule
$\leq$ maximum lateness in old schedule.
So we have not made max lateness any worse.

Correctness of Greedy Algorithm (7)

- Repeat procedure… like bubble sort.
  - We go down the optimal schedule doing swaps whenever an inversion is encountered.
    - We never increase the maximum lateness.

- At the end we have a schedule S' with no inversions and lateness as good as the original optimal schedule.

- Observation. S' is the schedule produced by greedy scheduling.

Points to Remember

- Greedy algorithms are usually simple to describe and have fast running times.
  - typically $\Theta(n \log n)$ or even $\Theta(n)$
- The challenge is to show that the greedy solution is indeed optimal.
  - One approach is to use induction, showing that any optimal solution can be modified to incorporate greedy choices without a loss of optimality.

Greedy Algorithms: Summary

- Characteristics of a greedy algorithm:
  - For it to work the solution to the problem should be constructible in an incremental fashion.
    - Typically, we may not get an optimal solution if there is a bad choice made at some particular step.
  - If the greedy algorithm works, it does so by doing the following at each step:
    - Using a cost function (also called a “weight” function), to evaluate all choices that can be made at a particular step.
    - Greedily make the choice that has the least cost or largest weight or largest gain.

Greedy Algorithms: Proof Template #1 (1)

“Entries in the optimal solution can be **sequentially replaced** by choices from the greedy solution”

1. Describe the greedy *local choice* strategy.
   Usually defined as some “locally optimal” choice.
2. Develop a representation for two solution descriptions:
   Describe the configuration of the greedy solution G.
   Describe the configuration of the optimal solution O.
3. Use an **inductive** argument to show that we can work through the optimal solution configuration and replace each of its entries with a choice made by the greedy algorithm. For *each* replacement:
   - Use the properties of G’s local optimal choice to show that such a replacement is possible without altering any other entry in the optimal solution.
   - Show further that the replacement will not harm the optimality of the O solution.

Greedy Algorithms: Proof Template #1 (2)

- The last two issues are applied to each of the following:
  - – Base case:
    - Show that an optimal solution can be constructed involving the first greedy choice.
  - – Induction step:
    - Assume that there is an optimal solution OPT that uses the greedy algorithm to generate the first k choices.
    - Building on this assumption, demonstrate that you can construct an optimal solution OPT' that has the same value as OPT (hence optimal) and such that it uses the greedy algorithm to generate the first $k + 1$ choices.

Greedy Algorithms: Proof Template #1 (3)

"Entries in the optimal solution can be **sequentially replaced** by choices from the greedy solution"

- These steps were seen in activity scheduling:

1. Describe the greedy *local choice* strategy.

   - Picking an activity with the **earliest** finish time.
2. Describe the configuration of the greedy solution: $G = \{g_1, g_2, g_3, \ldots, g_m\}$
   Describe the configuration of the optimal solution: $H = \{h_1, h_2, h_3, \ldots, h_q\}$
3. Use an **inductive** argument to show that we can work through the optimal solution configuration and replace each of its entries with a choice made by the greedy algorithm. For *each* replacement:
   - Use the properties of G's local optimal choice to show that such a replacement is possible without altering any other entry in the optimal solution.
     - – Based on properties of the finish times, we showed that replacement **maintained compatibility**.
   - Show further that the replacement will not harm the optimality of the O solution.
     - – We showed that the **number of activities did not change** when the optimal solution was modified to make it look like the greedy solution.

Greedy Algorithms: Proof Template #2 (1)

"Entries in the optimal solution can be systematically **exchanged** (swapped) to make the optimal solution look like the greedy solution"

1. Describe the greedy *local choice* strategy.
   Usually defined as some "locally optimal" choice.
2. Develop a representation for two solution descriptions:
   Describe the configuration of the greedy solution G.
   Describe the configuration of the optimal solution O.
3. Compare the greedy solution with the optimal solution to find where the greedy solution differs from the optimal solution.
4. Go through a constructive argument that works with exchanges to show that the optimal solution can be reordered to get an ordering like that in greedy while still maintaining optimality.
   - Usually demonstrated for one exchange followed by an argument that shows a complete conversion to the greedy configuration when the exchanges are repeated.

Greedy Algorithms: Proof Template #2 (2)

We saw this approach for greedy task scheduling:

1. Describe the greedy *local choice* strategy:
   Sort by deadline in increasing order (Earliest Deadline First = EDF).
2. Develop a representation for two solution descriptions:
   Describe the configuration of the greedy solution G.
   The schedule had $task(i)$ before $task(j)$ if: $d_i < d_j$.
   Describe the configuration of the optimal solution O.
   If O did not look like G then we argued that the optimal schedule had an inversion.
3. Compare the greedy solution with the optimal solution to find where the greedy solution differs from the optimal solution.
   An inversion in the optimal solution meant that we could find: $task(j)$ immediately before $task(i)$ if: $d_i < d_j$.
4. Go through a constructive argument that works with exchanges to show that the optimal solution can be reordered to get an ordering like that in greedy while still maintaining optimality.
   We used properties of EDF to show that an exchange does not increase the maximum lateness. REPEAT until all exchanges act like bubble sort to put the optimal solution into EDF (as used by greedy).