[Overall Information]

In this class, we studied the Interval Selection problem, which is the last Dynamic Programming problem we will cover in this course. The dynamic programming solution was derived step by step during the lecture. Its running time was also computed.

[Announcements]

1. The midterm exam is coming this Wednesday. It is an open-book exam. Besides the following items, please remember to bring a blue book as well.

   The materials you are allowed to bring for this open-book exam are:

   (1) The textbook;

   (2) Anything in your own writing;

   (3) The sample solutions of homework and quizzes.

   (4) Class notes provided by us, like this one.

2. There will be one review session on this Monday (03/20/2006) 3:30-4:50pm at THH116. You can either send the topics you’d like to hear in this session to the professor beforehand or come to the session with specific questions.
3. There will be extra office hours on Tuesday from 1:00 to 3:00pm at SAL232 (by the professor) and from 9:40am to 1:00pm as well as from 2:00 to 4:30pm at SAL311 (by the TA).

[During the Lecture]

1. In today’s lecture, we continued our study of Dynamic Programming. One more example was given.

The input of this Interval Selection problem is n intervals represented by $I_j = (l_j, r_j)$. Each interval has a weight $w_j$. Our goal is to find a collection of non-overlapping intervals such that the total weight is maximized.
Some real-life applications corresponding to this problem were talked about afterwards. They are: (1) optimizing one’s working schedule to maximize total salary, (2) finding the best way to utilize computer resources and (3) allocate network resource (on a single long link) for data transfer.

Several generalized versions of this problem were also briefly discussed, including what if we have multiple consultants / processors, some job may require more than one consultant to accomplish, etc.

Related to those problems, many topics are very interesting, and people
are doing active research on them.

3. To solve the problem, the first algorithm we had a look at is exhaustive search. For exhaustive search, the number of subsets we have to check is $2^n$. So the running time is very inefficient, $\Theta(2^n)$.

To improve the performance, we try to use greedy algorithms. The first greedy algorithm we may use is to always choose the non-overlapping jobs available with the highest payment. This greedy algorithm fails when you can take a longer job with payment 2, while at the same time interval, you can take several other shorter job with payment 1.
For the second greed algorithm we considered in the class, we always pick the jobs with best ratio $w_j/(r_j-l_j)$, here $w_j$ is the weight, or say, the money one can earn from the job, $(r_j-l_j)$ is the interval length, or say, the time needed to finish certain job. All of those greedy algorithms also fail when either we may choose a job with higher ratio and then have nothing to do during other time of the period, or we can choose a sequence of jobs with lower ratio but keep ourselves busy and gain a higher total amount of salaries.

4. Since there is always something wrong when using greedy algorithm to solve the Interval Selection problem, we came back to exhaustive search and tried to improve its running time performance, which is
frequently the first step towards deriving a Dynamic Program.

The search tree of exhaustive search reflecting the choice you make for each interval $I_j$ was drawn onto the whiteboard. The tree has $2^n$ leaf nodes. The pseudocode of exhaustive search was also provided.

5. Suppose we are running an exhaustive search (expressed as exhaustive (R)) and R represents the set of remaining intervals we have not considered yet.

If $R = \emptyset$ (empty set), we can gain nothing (no money) from it.

If $R \neq \emptyset$, let $j$ be an element in $R$, then we have:

(1) If we pick $j$ up, then all the intervals overlapping $j$ cannot be picked up afterwards. So the total gain from now on we will earn can be expressed as: $\text{exhaustive}(R - \{j\} - S_j) + w_j$, where $S_j$ is the set of all
intervals overlapping j and \( w_j \) is the weight of interval j.

(2) If j is not picked up, the total gain will become: \( \text{exhaustive}(R - \{j\}) \).

What the algorithm will choose is based on which one is larger between case (1) and case (2).

6. This is still inefficient (the running time in the worst case is still \( 2^n \), though some pruning that occurs may improve it in practice). In the spirit of dynamic programming, we may want to store \( a[R] \) (the maximum gain for subset R) somewhere and then reuse the value when \( \text{exhaustive}(R) \) is called instead of recomputing it from scratch.

Unfortunately, the number of subsets R is \( 2^n \). Thus, we have an algorithm with the same bad running time but requiring more memory. No progress yet.
7. We want to keep the same idea, but reduce the number of subsets $R$ for which we need to solve (and store) the solution. It will turn out that we can do that by first sorting all the intervals by right end point $r_j$ and start the yes/no branching from the rightmost remaining interval (suppose its index is $k$).

If it is not selected, the calculation goes to the next right most remaining interval prior to $k$, which will be $k-1$.

If it is selected, then all the intervals overlapping $k$ do not need to be computed afterwards, so we next go to some $k' < k$. 

This array contains an entry for each $R \in \{1, \ldots, n\}$, so has size $2^n$. So some running time, more memory.
Thus, we have derived the following properties of the optimum solution $OPT(k)$, which expresses the maximum gain that can be obtained solely from intervals with indices in $\{1,...,k\}$.

(1) $OPT(0) = 0$;

(2) $OPT(k) = \max\{OPT(k-1), OPT(\varphi(k)) + w_j\}$, where $\varphi(k)$ is the largest $j$ satisfying $r_j < l_k$ and $OPT(k)$ is the maximum gain we can have from jobs 1, 2, ..., $k$ alone. Notice that $\varphi(k)$ thus is exactly the largest index of any job that does not overlap interval $k$. 

If intervals are sorted by right endpoint $r_j$, and we start by branching on the rightmost remaining point $k$ ($k \in \Omega$)

(i) If $k$ is not selected, remaining intervals are $1, ..., k-1$.

(ii) If $k$ is selected, remaining intervals are $1, ..., k'$, where $k'$ is the largest index $j$ with $r_j < l_k$. 

The pseudocode of this dynamic programming solution was written on the whiteboard. The two precomputation steps before the searching begins are:

1. Sort $I_j$ by non-decreasing order of $r_j$. (running time: $\Theta(n \log(n))$ with MergeSort);

2. Pre-compute all $\varphi(j)$. (running time: $\Theta(n^2)$ with standard implementation, or $\Theta(n)$ slightly more sophisticated).

Since the subsequent loop takes $\Theta(n)$ time, the total running time is $\Theta(n \log(n))$, which is much better than exhaustive search.
DP solution

1. Sort $I_j$ by non-decreasing $y_j$.
2. Pre-compute all $y(j)$.
3. set $a(0):=0$;
4. for $j := 1$ to $n$ do
   $a(j):=\max(a[j-1], w_j + a[y(j)])$; \(\Theta(1)\)
5. return $a[n]$;

\[ T(n) = \Theta(n) \]

\[ \Theta(n \log n) \]