Lecture 6 & 7 Dynamic Programming Summary

**Dynamic Programming General Idea**: Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

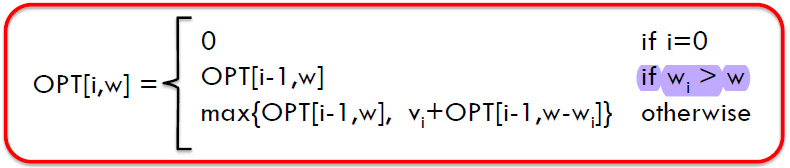
**Standard Correctness Proof:** Prove by Induction

**Dynamic programming**-> key steps:

**三个步骤**：

1. Define Sub-Problem
2. Find Recurrence
3. Solve Base Case

Dynamic Programming最后再写recurrence的时候需要把base case 和recurrence 写在一个大括号里。



**Weighted Interval Scheduling**:

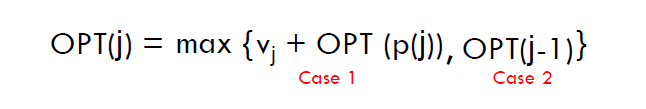
(Recall that unweighted interval scheduling problem is to find maximum subset of mutually compatible jobs, done by Greedy algorithm using Earliest Finish Time, which is the optimal solution, can be proved by Exchange Argument (Contradiction), with running time O(nlogn)).

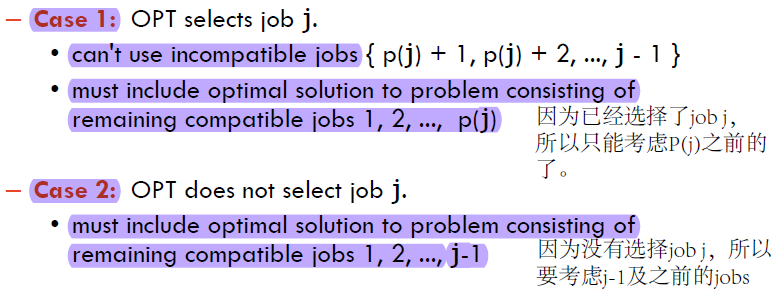
**Weighted interval scheduling**: Goal: to find the maximum weight subset of mutually compatible jobs.

**Compatible**: Two jobs are compatible if they don’t overlap.

**Prior function**: P(j) = largest index i < j such that job i is compatible with j. (The largest index of the un-overlapped job before job i)

* **Sub-Problem**: OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, .., j.
* **Recurrence**: OPT(j) = max {Vj+OPT(p(j)) , OPT(j-1)}, while the first one corresponds to select job j, while the second one corresponding to not select job j.





* **Base Case**: OPT(0) = 0

**Proof of the Dynamic Programming Algorithm**: **Proof by Induction**.

先说base case is correct, inductive hypothesis 当index < k时全是correct，之后inductive step证明k是correct。 Done。

**Running time of weighted interval scheduling**: O(nlogn)->sorting takes O(nlogn), computing p() takes O(n). Running time is O(n) if the jobs are pre-sorted by start and finish times.

**Dynamic programming的时间复杂度**

如果使用memoization 存储所有先前计算结果的话，时间复杂度为O(n) if jobs are pre-sorted by start and finish time. Memoized version of the algorithm takes O(n\*logn) time.

Memoization: Store results of each sun-problem, lookup when needed.

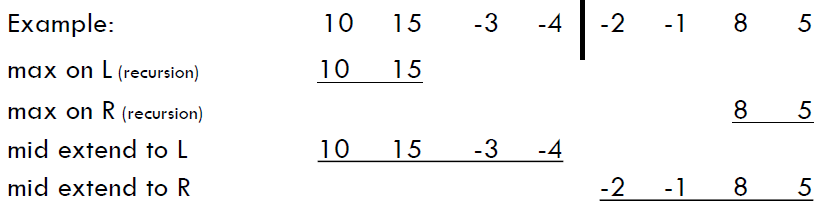
**顺序**

**Bottom-up dynamic programming**->iteratively

**Top-down dynamic programming**->Recursively.

**Maximum Sum Contiguous Sub-Array using Divide and Conquer**

1. Find the max on LHS
2. Find the max on RHS
3. Extend LHS and RHS to middle
4. Find the max of all these three possibilities.



Running time O(n\*logn).

**Maximum Sum Contiguous Sub-Array**

Maximum Sum Contiguous Sub-Array with time complexity O(n) and Space Complexity O(n).

**Sub-Problem**: OPT(i) = optimal solution ending at i

**Recurrence**: OPT(i) = max(OPT(i-1) + array(i) , 0) --> If the sum is less than 0, write it as 0, instead of a negative number.

**Base case** = OPT(max(array(1) , 0)).

**Knapsack Problem** (Each job has weight w and value v, knapsack has capacity W, the goal is to fill knapsack so as to maximise total value.)

Time Complexity O(nW) , Space Complexity O(nW).

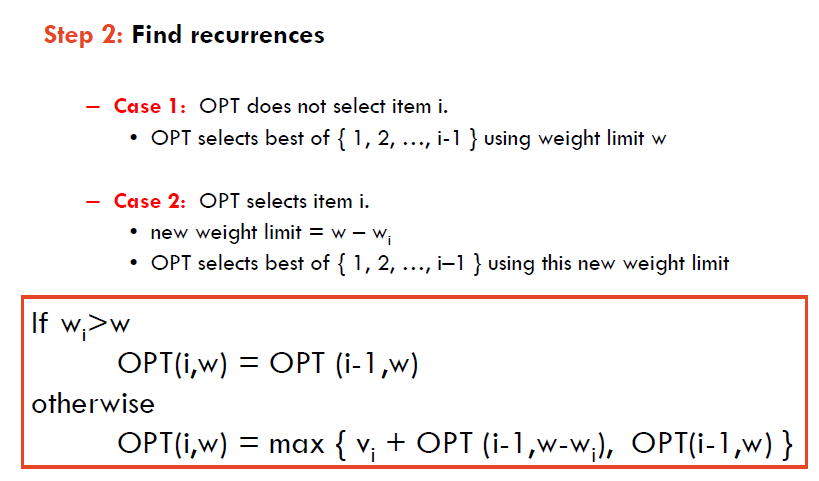
Decision version of Knapsack is NP-complete.

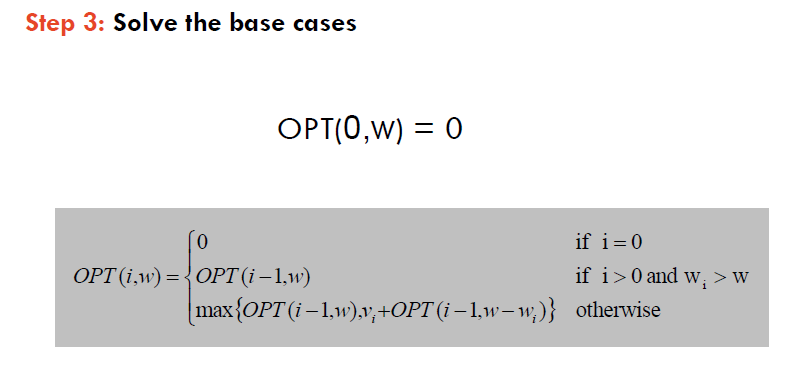
**Steps of Knapsack Problem**

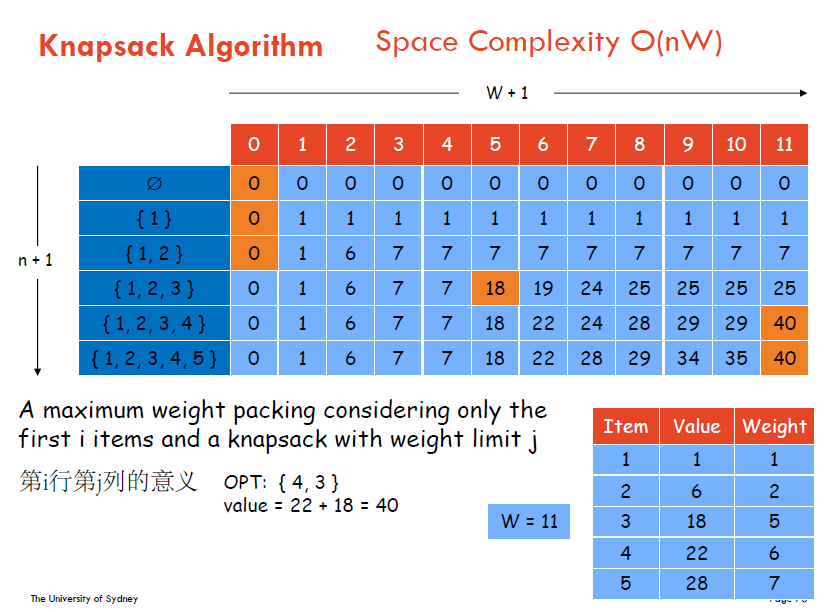
Sub-Problem: OPT[I , w] = max profit subset of items 1 , … , i with weight limit w.

Recurrence: OPT[I , w] = max{OPT[i-1 , w] , value(i) + OPT[i-1 , w-wi]}

Base Case: OPT[0 , w] = 0;





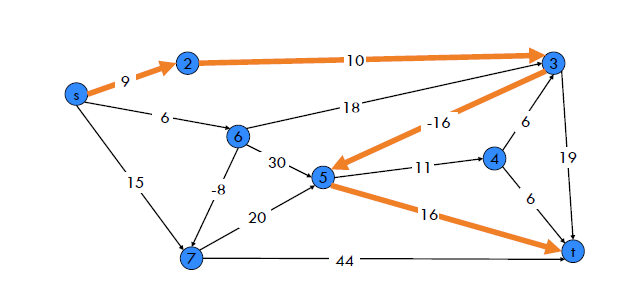


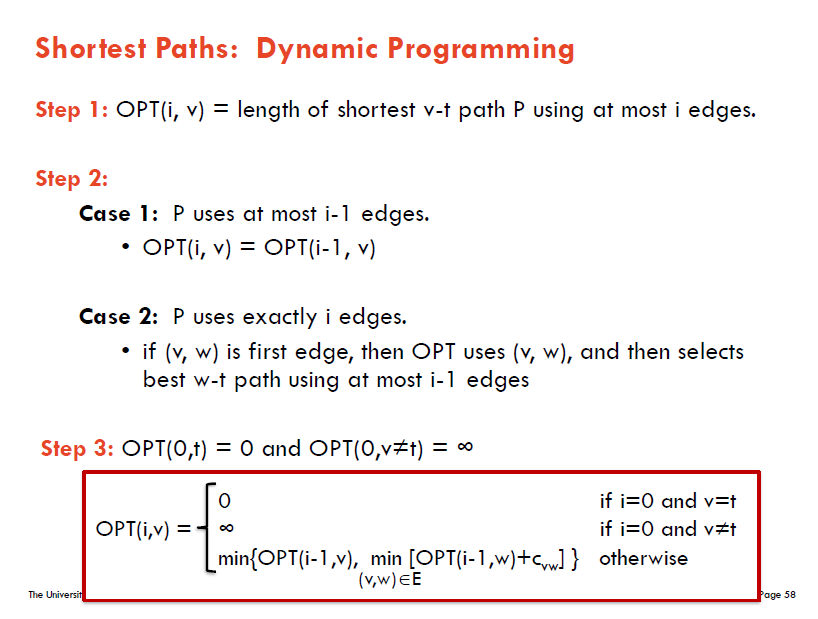
**Knapsack Problem Traceback** to find out which item we pick: If we are at point (x, y) in this table, which has value v, we look at the value above it, which is point (x, y-1). If it is the same as v, we didn’t use item y, so we look at (x, y-1) next. If it is different from value v, we did use item y. If item y has weight w, we now look at point (x-w , y-1) net. This continues until we know whether we used each item or not.

**Bellman Ford Algorithm** (Shortest Path) (Solving using Dynamic Programming) (Allow negative weights)

**Idea**: The Bellman-Ford Algorithm correctly computes the shortest path in a directed graph even when the graph contains edges which have negative weights. The algorithm has time complexity O(nm) and space complexity O(n+m), where n is the number of vertices and m is the number of edges.

**Observation**: If some path from S to T contains a negative cost cycle, there does not exist a shortest S-T path; otherwise, there exists one that is simple.





Dijkstra’s Algorithm can fail if there exists negative edge cost.

**Define Sub-Problems**: OPT(I , v) = length of shortest v-t path P using at most I edges.

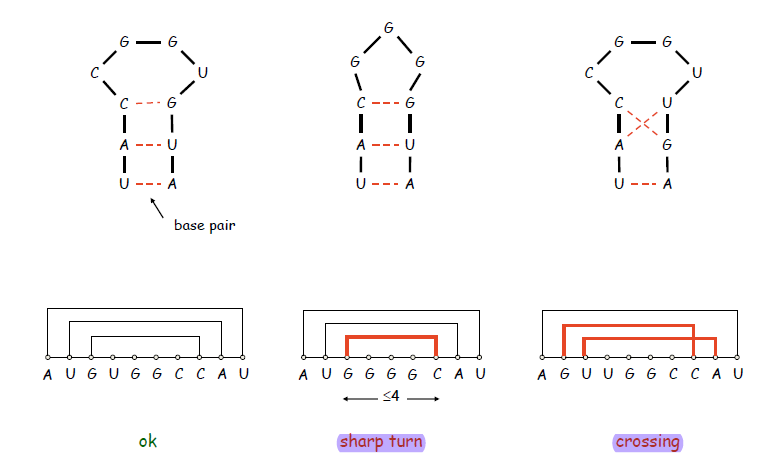
**Recurrence** Relationship of Bellman Ford:

OPT(I , v) = min{OPT(i-1 , v) , min(OPT(i-1 , w) + Cvw)}, where the first one means the path will use exactly i-1 edges, and will not use any edges between v and w, while the second one means the algorithm will use the smallest cost edge between v and w, and will also use another i-1 edges.

**Base Case**: OPT(0 , t) = 0 ; OPT(0 , v!=t) = infinity.

**RNA Secondary Structure**:

* Watson-Click Rule: A-U U-A C-G G-C
* No Sharp Turns: The ends of each pair are separated by AT LEAST 4 intervening bases.
* Non-Crossing: Two pairs cannot crossing each other.



**Goal**: Given an RNA molecule, find a secondary structure S that maximize the number of base pairs.

**Sub-Problems**: OPT(I , j) = maximum number of base pairs in a secondary structure of the substring bi to bj.

**Recurrence**: OPT(I , j) = max{OPT(i , j-1) , 1+max{OPT(i , t-1) + opt(t+1 , j-1)} }, where i<=t<j-4.

The first case is base bj not involved in a pair, while the second case is base bj pairs with bt for some i<=t<j-4.

**Base Case**: if i>=j-4, then OPT(I , j) = 0 by Non-Sharp Turn condition.

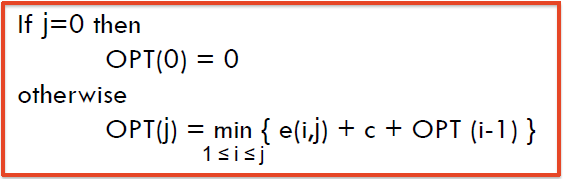
Time Complexity O(n3)

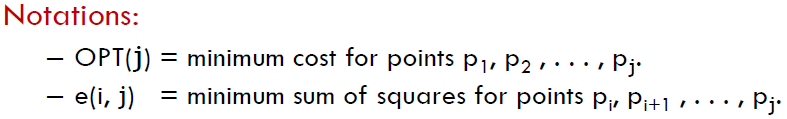
Space Complexity O(n2)

**Segmented Least Square**

**Sub-Problem**: OPT(j) = minimum cost for points p1 to pj

**Recurrence**

****

****

**Base Case:** OPT(0) = 0

Running time O(n3), space complexity O(n2).

1D Dynamic Programming:

1. Weighted interval scheduling
2. Segmented least square
3. Maximum-sum contiguous sub-array
4. Longest increasing subsequence

2D Dynamic Programming:

1. Knapsack
2. Shortest Path

Dynamic Programming over Intervals:

1. RNA Secondary Structure.

**Longest Increasing Subsequence**

Running time O(nlogn)