Weighted Activity Selection Activity Partitioning

CSCI 3100

Review & Overview

Last time

- Introduced activity selection problem
- Formulated solution in terms of sub-problems
- Solved using greedy approach
- Greedy strategy: pick activity with earliest finish time

Today

- Weighted version of activity selection problem
- Why 'earliest finish time' greedy strategy does not work
- Solution
- Activity partitioning problem

Weighted activity selection

Given a set of activities S = {a_1, a_2, …, a_n}, where each activity is a 3-tuple with (start, finish, value), select a subset of non-overlapping activities such that the total value of this subset is maximized.

- Activities are sorted in ascending order by finish time
- $finish_1 \leq finish_2 \leq \ldots \leq finish_n$

Non-weighted version of activity selection can be turned into weighted, by assigning each activity a weight of 1.

Earliest Finish Time greedy approach does not work

Come up with an example where the greedy approach of scheduling the activity with the earliest finish time will not produce an optimal solution

Optimal substructure

Let S_j be a subset of S with the first j activities

Let opt[S_j] be the maximum value that can be obtained from scheduling activities in S_j

opt[S_j] is the larger of the two values:

- Profit obtained by excluding job j in the solution (opt[S_{j-1}])
- Profit obtained by including job j in the solution (value[j] + opt[A_j])
  - where $A_j \subset S_j$, contains activities that do not overlap with a_j

opt[S_j] = max{ opt[S_{j-1}], value[j] + opt[A_j])

Divide & Conquer – Recursive solution

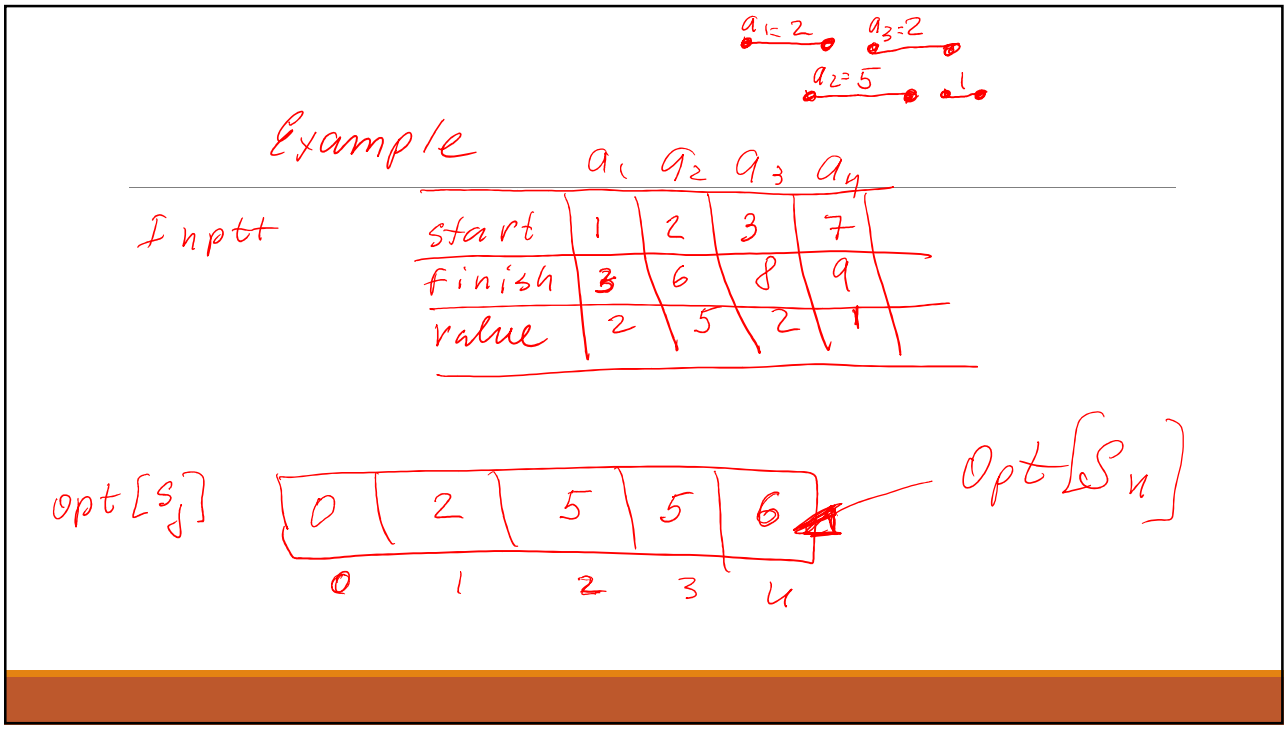
opt[S_j] = max{ opt[S_{j-1}], value[j] + opt[A_j] }

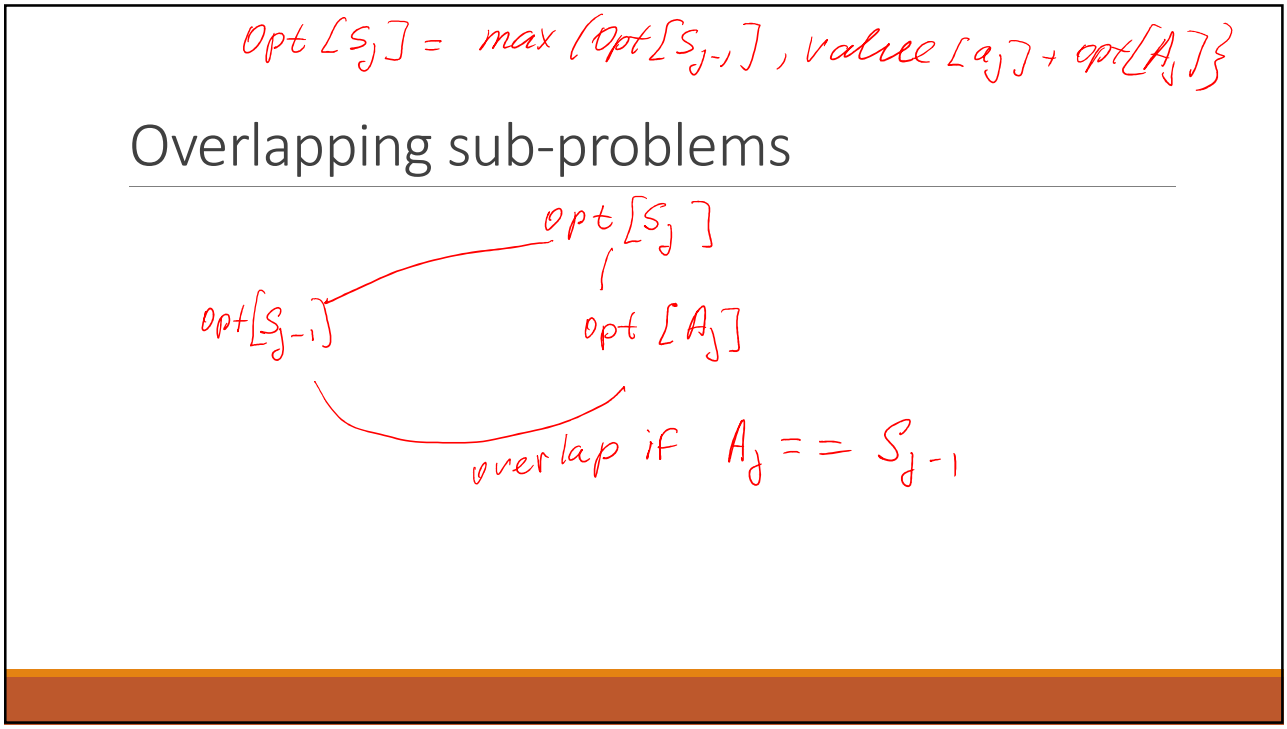
opt[S_{j-1}] : excl value

value[j] + opt[A_j] : incl value

```
findMaxValueRec(a, n)
{
    if (n == 1) return a[1].value;
    inclValue = a[n].value;
    i - index of last activity that does not conflict with a[n]
    if (i != -1)
        inclValue += findMaxValueRec(a, i);
    exclValue = findMaxValueRec(a, n-1);
    return max(inclValue, exclValue)
}
```

a[i].start
a[i].finish
a[i].value
a is sorted by finish time


a1=2
a3=2
a2=5
Example
a1 a2 a3 a4
Input
start 1 2 3 7
finish 3 6 8 9
value 2 5 2 1
opt[Sj]
0 2 5 5 6
Opt[S4]
0 1 2 3 4


Opt[Sj] = max(Opt[Sj-1], value[aj] + opt[Aj])
Overlapping sub-problems
opt[Sj]
opt[Sj-1]
opt[Aj]
overlap if Aj == Sj-1

$Opt[S_j] = \max\{opt[S_{j-1}], value[a_j] + opt[A_j]\}$

Divide & Conquer – Dynamic Programming Solution

$\Theta(n^2)$

```
findMaxValue (a, n)
{   value - array of size n
    value[0] = 0
    value[1] = a[1].value
    for (j = 2; j ≤ n; j++)
    {   exclValue = value[j-1]
        i - latest index of activity not conflicting
            with a[j]
        if (i ≠ -1)
            inclValue = a[j].value + value[i]
        value[j] = max(inclValue, exclValue);
    }
return value[n]
```

Find latest compatible activity

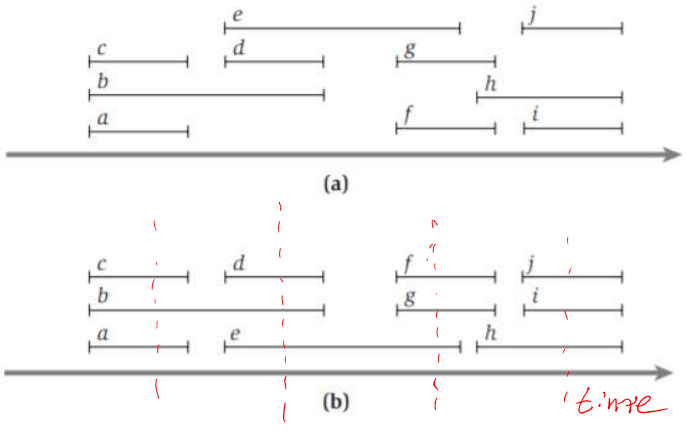
```
lastCompatibleActivity (a, j)
{
    for (i = j-1; i>0; i--)
    {   if (a[i].finish ≤ a[j].start)
            return i;
    }
    return -1;
}
```

Activity

Interval Partitioning Problem

Given: set of n lectures with start and finish times

Goal: schedule the lectures using minimum number of classrooms, so that no two lectures are scheduled in the same classroom at the same time



Example

- Figure (a) and Figure (b) shows two solutions to the same problem
- Solution in Figure (a) uses 4 classrooms
- Solution in Figure (b) uses 3 classrooms
- Can we do better than 3 classrooms?
- Depth of the set of intervals – max number that pass over any single point at the same time
- In the example, depth = 3
- If we use # of classrooms equal to the *depth*, this solution is optimal

Algorithm sketch

Sort lectures in increasing order by start time

Assign next lecture to some classroom k, if it is compatible with k

If lecture is not compatible with any classrooms c1, …, ck, open a new classroom k+1

Proof of optimality:

Algorithm

```
1:  function SCHEDULE(s_1, s_2, ..., s_n)
2:      Sort the intervals by starting time in non-decreasing order.
3:      d ← 0
4:      for j = 1 to n do
5:          if lecture j is compatible with some classroom k then
6:              schedule lecture j in classroom k
7:          else
8:              allocate a new classroom d + 1
9:              schedule lecture j in the new classroom
10:             d ← d + 1
```

Time complexity: