weighted interval scheduling

• Input. Set of jobs with start times, finish times, and weights.

• Goal. Find maximum weight subset of mutually compatible jobs.

step 1 – a recursive algorithm

• All subproblems involve requests \{1, ..., i\} for some i

• For i=1,...,n let OPT(i) be the weight of the optimal solution to the problem \{1, ..., i\}

• The two cases give

  \[ OPT(n) = \max\{w_i + OPT(p(n)), OPT(n-1)\} \]

  \[ n \in \mathcal{O} \text{ iff } w_n + OPT(p(n)) > OPT(n-1) \]
step 1 – a recursive algorithm

First, sort requests and compute array $p[i]$ for each $i = 1, \ldots, n$.

\[
\text{ComputeOpt}(n)
\]
\[
\quad \text{if } n = 0 \text{ then return}(0)
\]
\[
\quad \text{else}
\]
\[
\quad \quad u \leftarrow \text{ComputeOpt}(p[n])
\]
\[
\quad \quad v \leftarrow \text{ComputeOpt}(n-1)
\]
\[
\quad \quad \text{if } w_n + u > v \text{ then}
\]
\[
\quad \quad \quad \text{return}(w_n + u)
\]
\[
\quad \quad \text{else}
\]
\[
\quad \quad \quad \text{return}(v)
\]
\[
\quad \text{endif}
\]

step 2 – memoization

- \text{ComputeOpt}(n) can take exponential time in the worst case
  - $2^n$ calls if $p(i) = i - 1$ for every $i$
- There are only $n$ possible parameters to \text{ComputeOpt}
- Store these answers in an array $OPT[n]$ and only recompute when necessary
  - Memoization
- Initialize $OPT[i] = 0$ for $i = 1, \ldots, n$

step 2 – memoization

\[
\text{ComputeOpt}(n):
\]
\[
\quad \text{if } n = 0 \text{ then return}(0)
\]
\[
\quad \text{else}
\]
\[
\quad \quad u \leftarrow \text{MComputeOpt}(p[n])
\]
\[
\quad \quad v \leftarrow \text{MComputeOpt}(n-1)
\]
\[
\quad \quad \text{if } w_n + u > v \text{ then}
\]
\[
\quad \quad \quad \text{return}(w_n + u)
\]
\[
\quad \quad \text{else}
\]
\[
\quad \quad \quad \text{return}(v)
\]
\[
\quad \text{endif}
\]

MComputeOpt(n):

\[
\quad \text{if } OPT[n] = 0 \text{ then}
\]
\[
\quad \quad v \leftarrow \text{ComputeOpt}(n)
\]
\[
\quad \quad OPT[n] \leftarrow v
\]
\[
\quad \text{return}(v)
\]
\[
\quad \text{else}
\]
\[
\quad \quad \text{return}(OPT[n])
\]
\[
\quad \text{endif}
\]

step 3 – iterative solution

The recursive calls for parameter $n$ have parameter values $i$ that are $< n$.
step 3 – iterative solution

The recursive calls for parameter \( n \) have parameter values \( i \) that are < \( n \)

\[
\text{IterativeComputeOpt}(n) \\
\text{array OPT[0, \ldots, n]} \\
\text{OPT[0]} \leftarrow 0 \\
\text{for } i = 1 \text{ to } n \\
\quad \text{if } w_i + \text{OPT}[p[i]] > \text{OPT}[i-1] \text{ then} \\
\quad \quad \text{OPT}[i] \leftarrow w_i + \text{OPT}[p[i]] \\
\quad \text{else} \\
\quad \quad \text{OPT}[i] \leftarrow \text{OPT}[i-1] \\
\quad \text{endif} \\
\text{endfor} \\
\]

producing an optimal solution

\[
\text{IterativeComputeOptSolution}(n) \\
\text{array OPT[0, \ldots, n], Used[1, \ldots, n]} \\
\text{OPT[0]} \leftarrow 0 \\
\text{for } i = 1 \text{ to } n \\
\quad \text{if } w_i + \text{OPT}[p[i]] > \text{OPT}[i-1] \text{ then} \\
\quad \quad \text{OPT}[i] \leftarrow w_i + \text{OPT}[p[i]] \\
\quad \quad \text{Used[i]} \leftarrow 1 \\
\quad \text{else} \\
\quad \quad \text{OPT}[i] \leftarrow \text{OPT}[i-1] \\
\quad \text{Used[i]} \leftarrow 0 \\
\quad \text{endif} \\
\text{endfor} \\
\]

producing an optimal solution

\[
\text{IterativeComputeOptSolution}(n) \\
\text{array OPT[0, \ldots, n], Used[1, \ldots, n]} \\
\text{OPT[0]} \leftarrow 0 \\
\text{for } i = 1 \text{ to } n \\
\quad \text{if } w_i + \text{OPT}[p[i]] > \text{OPT}[i-1] \text{ then} \\
\quad \quad \text{OPT}[i] \leftarrow w_i + \text{OPT}[p[i]] \\
\quad \quad \text{Used[i]} \leftarrow 1 \\
\quad \text{else} \\
\quad \quad \text{OPT}[i] \leftarrow \text{OPT}[i-1] \\
\quad \text{Used[i]} \leftarrow 0 \\
\quad \text{endif} \\
\text{endfor} \\
\]

example

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
s_i & 4 & 2 & 6 & 8 & 11 & 15 & 11 & 12 & 18 \\
\hline
f_i & 7 & 9 & 10 & 13 & 14 & 17 & 18 & 19 & 20 \\
\hline
w_i & 3 & 7 & 4 & 5 & 3 & 2 & 7 & 7 & 2 \\
\hline
p[i] & & & & & & & & & \\
\hline
\text{OPT}[i] & & & & & & & & & \\
\hline
\text{Used}[i] & & & & & & & & & \\
\end{array}
\]
example

### Least Squares

- Given a set $P$ of $n$ points in the plane $\mathbf{p}_1=(x_1, y_1), \ldots, \mathbf{p}_n=(x_n, y_n)$ with $x_1 < \ldots < x_n$ determine a line $L$ given by $y=Ax+B$ that optimizes the totaled ‘squared error’

$$\text{Error}(L, P) = \sum_i (y_i - Ax_i - B)^2$$

- A classic problem in statistics

- Optimal solution is known (see text)

Call this line($P$) and its error error($P$)

---

S={9,7,2}
What if data seems to follow a piece-wise linear model?
What if data seems to follow a piece-wise linear model?
• Number of pieces to choose is not obvious
• If we chose \( n-1 \) pieces we could fit with 0 error
  – Not fair
• Add a penalty of \( C \) times the number of pieces to the error to get a total penalty
• How do we compute a solution with the smallest possible total penalty?

Recursive idea
– If we knew the point \( p_j \) where the last line segment began then we could solve the problem optimally for points \( p_1,...,p_j \) and combine that with the last segment to get a global optimal solution
Let \( OPT(i) \) be the optimal penalty for points \( \{p_j,...,p_i\} \)
Total penalty for this solution would be
\[
\text{Error}\{p_j,...,p_n\} + C + OPT(j-1)
\]

Recursive idea
– We don’t know which point is \( p_j \)
  But we do know that \( 1 \leq j \leq n \)
  The optimal choice will simply be the best among these possibilities
– Therefore:
segmented least squares

Recursive idea
– We don’t know which point is \( p_j \)
  But we do know that \( 1 \leq j \leq n \)
  The optimal choice will simply be the best among these possibilities
– Therefore:

\[
\text{OPT}(n) = \min_{1 \leq j \leq n} \{ \text{Error}(\{ p_j, \ldots, p_n \}) + C + \text{OPT}(j - 1) \}
\]

dynamic programming solution

\[
\text{SegmentedLeastSquares}(n) \quad \text{array OPT[0,...,n]}, \text{Begin[1,...,n]}
\]
\[
\text{OPT}[0] \leftarrow 0
\]
for \( i=1 \) to \( n \)
\[
\text{OPT}[i] \leftarrow \text{Error}(\{ p_1, \ldots, p_i \}) + C
\]
\[
\text{Begin}[i] \leftarrow 1
\]
for \( j=2 \) to \( i-1 \)
\[
\text{e} \leftarrow \text{Error}(\{ p_j, \ldots, p_i \}) + C + \text{OPT}[j-1]
\]
if \( e < \text{OPT}[i] \) then
\[
\text{OPT}[i] \leftarrow e
\]
\[
\text{Begin}[i] \leftarrow j
\]
endif
endfor
endfor
return(\text{OPT}[n])

knapsack (subset-sum) problem

• Given:
  – integer \( W \) (knapsack size)
  – \( n \) object sizes \( x_1, x_2, \ldots, x_n \)

• Find:
  – Subset \( S \) of \( \{1, \ldots, n\} \) such that \( \sum_{i \in S} x_i \leq W \)
    but \( \sum_{i \in S} x_i \) is as large as possible

recursive algorithm

• Let \( K(n,W) \) denote the problem to solve for \( W \) and \( x_1, x_2, \ldots, x_n \)
recursive algorithm

- Let $K(n, W)$ denote the problem to solve for $W$ and $x_1, x_2, \ldots, x_n$

- For $n > 0$,
  - The optimal solution for $K(n, W)$ is the better of the optimal solution for either $K(n-1, W)$ or $x_n + K(n-1, W-x_n)$
  - For $n=0$
    $K(0, W)$ has a trivial solution of an empty set $S$ with weight 0

common sub-problems

- Only sub-problems are $K(i, w)$ for
  - $i = 0, 1, \ldots, n$
  - $w = 0, 1, \ldots, W$

- Dynamic programming solution
  - Table entry for each $K(i, w)$
    $OPT$ - value of optimal soln for first $i$ objects and weight $w$
    $belong$ flag - is $x_i$ a part of this solution?
  - Initialize $OPT[0, w]$ for $w=0, \ldots, W$
  - Compute all $OPT[i, *]$ from $OPT[i-1, *]$ for $i>0$

recursive calls

Recursive calls on list ..., 3, 4, 7

dynamic knapsack algorithm

```plaintext
time O(nW)
```

```plaintext
for w=0 to W: OPT[0, w] ← 0; end for
for i=1 to n do
  for w=0 to W do
    OPT[i, w] ← OPT[i-1, w]
    belong[i, w] ← 0
    if $w \geq x_i$ then
      val ← $x_i + OPT[i-1, w-x_i]$
      if val > OPT[i, w] then
        OPT[i, w] ← val
        belong[i, w] ← 1
      end for
  end for
end for
return(OPT[n, W])
```
sample execution on 2, 3, 4, 7 with W=15

saving space

• To compute the value $OPT$ of the solution only need to keep the last two rows of $OPT$ at each step

• What about determining the set $S$?
  – Follow the $belong$ flags $O(n)$ time
  – What about space?

three steps to dynamic programming

• Formulate the answer as a recurrence relation or recursive algorithm

• Show that the number of different values of parameters in the recursive algorithm is “small”
  – e.g., bounded by a low-degree polynomial

• Specify an order of evaluation for the recurrence so that you already have the partial results ready when you need them.

RNA secondary structure

• RNA: sequence of bases
  – String over alphabet \{A, C, G, U\}


• RNA folds and sticks to itself like a zipper
  – A bonds to U
  – C bonds to G
  – Bends can’t be sharp
  – No twisting or criss-crossing

• How the bonds line up is called the RNA secondary structure
RNA secondary structure

- **Input**: String $x_1...x_n \in \{A,C,G,U\}^*$
- **Output**: Maximum size set $S$ of pairs $(i,j)$ such that
  - $\{x_i,x_j\} = \{A,U\}$ or $\{x_i,x_j\} = \{C,G\}$
  - The pairs in $S$ form a matching
  - $|j-i| \leq 4$ (no sharp bends)
  - No crossing pairs
    - If $(i,j)$ and $(k,l)$ are in $S$ then it is not the case that they cross as in $i<k<j<l$

another view

recursive solution

Try all possible matches for the last base

General form:

$$OPT(i..j) = \max(\max(\text{OPT}(i..j-1), 1+\text{OPT}(i..k-1)+\text{OPT}(k+1..j-1)), \text{OPT}(i..k-1)+\text{OPT}(k+1..j-1)))$$

$x_k$ matches $x_j$ Doesn’t start at 1
RNA secondary structure

- 2D Array $OPT(i,j)$ for $i ≤ j$ represents optimal # of matches entirely for segment $i..j$
- For $j-i ≤ 4$ set $OPT(i,j)=0$ (no sharp bends)
- Then compute $OPT(i,j)$ values when $j-i=5,6,...,n-1$ in turn using recurrence.
- Return $OPT(1,n)$
- Total of $O(n^2)$ time
- Can also record matches along the way to produce S
  - Algorithm is similar to the polynomial-time algorithm for Context-Free Languages based on Chomsky Normal Form from 322
  - Both use dynamic programming over intervals