weighted interval scheduling

- **Input.** Set of jobs with start times, finish times, and weights.
- **Goal.** Find maximum weight subset of mutually compatible jobs.

---

**greedy algorithm?**

No criterion seems to work

- Earliest start time $s_i$
  - Doesn't work
  - Doesn't work

- Shortest request time $f_i - s_i$
  - Doesn't work

- Fewest conflicts
  - Doesn't work

- Earliest finish time $f_i$
  - Doesn't work

- Largest weight $w_i$
  - Doesn't work

---

**dynamic programming**

**Dynamic Programming**

- Give a solution of a problem using smaller sub-problems where the parameters of all the possible sub-problems are determined in advance

- Useful when the same sub-problems show up again and again in the solution
computing fibonacci numbers

- Recall \( F_n = F_{n-1} + F_{n-2} \) and \( F_0 = 0, F_1 = 1 \)
- Recursive algorithm:

full call tree

memoization (caching)

- Remember all values from previous recursive calls
- Before recursive call, test to see if value has already been computed
- **Dynamic Programming**
  - Convert memoized algorithm from a recursive one to an iterative one
finboacci: dynamic programming

FiboDP(n):
F[0] ← 0
F[1] ← 1
for i = 2 to n do
    F[i] ← F[i-1] + F[i-2]
endfor
return(F[n])

dynamic programming

Useful when:
- Same recursive sub-problems occur repeatedly
- Can anticipate the parameters of these recursive calls
- The solution to whole problem can be figured out with knowing the internal details of how the sub-problems are solved

principle of optimality:
"Optimal solutions to the sub-problems suffice for optimal solution to the whole problem"

fibonacci: space saving dynamic program

FiboDP(n):
    prev ← 0
    curr ← 1
for i = 2 to n do
    temp ← curr
    curr ← curr + prev
    prev ← temp
endfor
return(curr)
	hree 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 calls is "small"
  - e.g., bounded by a low-degree polynomial
  - Can use memoization
- Specify an order of evaluation for the recurrence so that you already have the partial results ready when you need them.
weighted interval scheduling

- **Input.** Set of jobs with start times, finish times, and weights.
- **Goal.** Find maximum weight subset of mutually compatible jobs.

![Diagram](https://via.placeholder.com/150)

step 1 – a recursive algorithm

Two cases depending on whether an optimal solution \( \mathcal{O} \) includes request \( n \)

- If it **does** include request \( n \) ...

step 1 – a recursive algorithm

Two cases depending on whether an optimal solution \( \mathcal{O} \) includes request \( n \)

- If it **does** include request \( n \) then all other requests in \( \mathcal{O} \) must be contained in \( \{1, \ldots, p(n)\} \)
  - Not only that!
  - Any set of requests in \( \{1, \ldots, p(n)\} \) will be compatible with request \( n \)
  - So in this case the optimal solution \( \mathcal{O} \) must contain an optimal solution for \( \{1, \ldots, p(n)\} \)
  - “Principle of Optimality”
step 1 – a recursive algorithm

Two cases depending on whether an optimal solution \( O \) includes request \( n \)

– If it \textbf{does} include request \( n \) ...

• All subproblems involve requests \( \{1, \ldots, l \} \) for some \( l \)

• For \( l=1, \ldots, n \) let \( \text{OPT}(l) \) be the \textit{weight} of the optimal solution to the problem \( \{1, \ldots, l \} \)

• The two cases give:

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

• Also \( n \in O \) \textbf{iff} \( w_n + \text{OPT}(p(n)) > \text{OPT}(n-1) \)

step 1 – a recursive algorithm

Two cases depending on whether an optimal solution \( O \) includes request \( n \)

– If it \textbf{does not} include request \( n \) then all requests in \( O \) must be contained in \( \{1, \ldots, n-1\} \)

Not only that!

The optimal solution \( O \) must contain an optimal solution for \( \{1, \ldots, n-1\} \)

“Principle of Optimality”
step 1 – a recursive algorithm

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

```plaintext
ComputeOpt(n):
  if $n = 0$ then return(0)
  else
    $u \leftarrow$ ComputeOpt($p[n]$)
    $v \leftarrow$ ComputeOpt($n-1$)
    if $w_n + u > v$ then
      return($w_n + u$
    else
      return($v$
  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 $\text{OPT}[n]$ and only recompute when necessary
  - Memoization
- Initialize $\text{OPT}[l] = 0$ for $l = 1, ..., n$

```plaintext
MComputeOpt(n):
  if $\text{OPT}[n] = 0$ then
    $v \leftarrow \text{ComputeOpt}(n)$
    $\text{OPT}[n] \leftarrow v$
    return(v)
  else
    return($\text{OPT}[n]$)
  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 < $n$

IterativeComputeOpt($n$)
array $OPT[0,...,n]$
$OPT[0] \leftarrow 0$
for $i=1$ to $n$
  if $w_i + OPT[p[i]] > OPT[i-1]$ then
    $OPT[i] \leftarrow w_i + OPT[p[i]]$
  else
    $OPT[i] \leftarrow OPT[i-1]$
  endif
endfor

producing an optimal solution

producing an optimal solution

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

example

<table>
<thead>
<tr>
<th>$s_i$</th>
<th>4</th>
<th>2</th>
<th>6</th>
<th>8</th>
<th>11</th>
<th>15</th>
<th>11</th>
<th>12</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_i$</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>13</td>
<td>14</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>$w_i$</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>$p[i]$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$OPT[i]$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Used[i]$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Least Squares

- **Definition:**
  - Given a set \( P \) of \( n \) points in the plane
  - \( P = (x_1, y_1), \ldots, (x_n, y_n) \) with \( x_1 < \ldots < x_n \) determine a line \( L \) given by \( y = ax + b \) that optimizes the totaled ‘squared error’
  - \( \text{Error}(L, P) = \sum_i (y_i - ax_i - b)^2 \)

- **Problem:**
  - A classic problem in statistics
  - Optimal solution is known (see text)
  - Call this line \( P \) and its error \( \text{error}(P) \)

### Segmented Least Squares

- Example:
  - Consider the points
  - \( S = \{9, 7, 2\} \)
  - Use segmented least squares to minimize the squared error.

#### Table 1:

<table>
<thead>
<tr>
<th>( s_i )</th>
<th>( f_i )</th>
<th>( w_i )</th>
<th>( p[i] )</th>
<th>( \text{OPT}[i] )</th>
<th>( \text{Used}[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>15</td>
</tr>
</tbody>
</table>

#### Table 2:

<table>
<thead>
<tr>
<th>( s_i )</th>
<th>( f_i )</th>
<th>( w_i )</th>
<th>( p[i] )</th>
<th>( \text{OPT}[i] )</th>
<th>( \text{Used}[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>9</td>
<td>10</td>
<td>13</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

#### Table 3:

<table>
<thead>
<tr>
<th>( s_i )</th>
<th>( f_i )</th>
<th>( w_i )</th>
<th>( p[i] )</th>
<th>( \text{OPT}[i] )</th>
<th>( \text{Used}[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>15</td>
</tr>
</tbody>
</table>

---

*Note:*

- The tables represent different sets of points and their associated values.
- The segmented least squares approach is used to minimize the squared error.
- The optimal solution is determined by minimizing the error function.

---

*Example:* Consider the points \( S = \{9, 7, 2\} \), use segmented least squares to minimize the squared error.
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 \( \text{OPT}(i) \) be the optimal penalty for points \( \{p_j,...,p_i\} \)

Total penalty for this solution would be

\[
\text{Error}(\{p_j,...,p_n\}) + C + \text{OPT}(j-1)
\]

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:

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

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