CSE 421: Algorithms

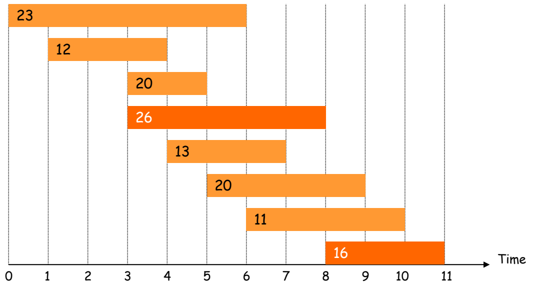
Winter 2014

Lecture 13: Dynamic programming

Reading:
Sections 6.1-6.3

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

- Shortest request time f_i-s_i
  Doesn't work

- Fewest conflicts
  Doesn't work

- Earliest finish fime f_i
  Doesn't work

- Largest weight w_i
  Doesn't work

dynamic programming

Dynamic Programming

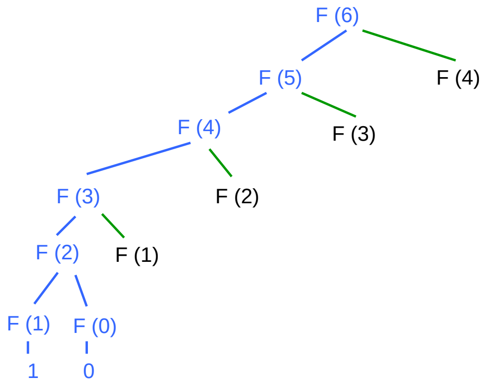
- Give a solution of a problem using smaller sub-problems where the parameters of all the possible sub-problems are determined in advance

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

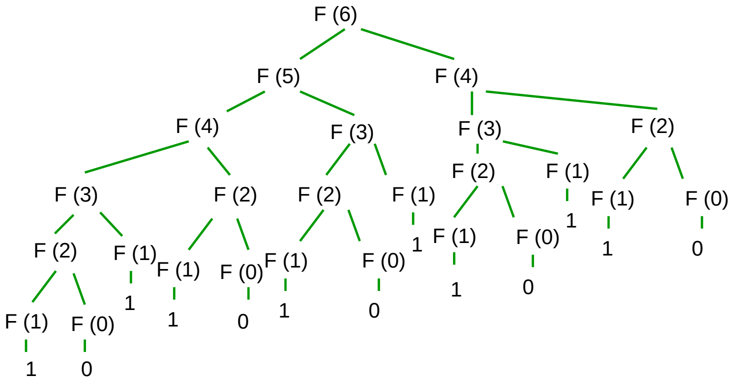
computing fibonaci numbers

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

call tree



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])
```

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)
```

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:

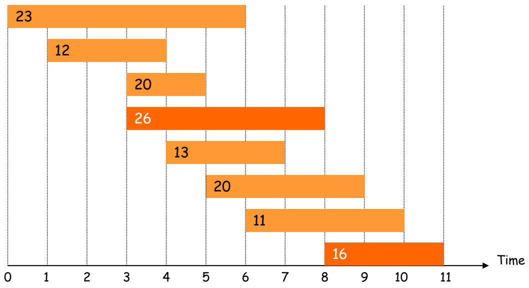
  "Optimal solutions to the sub-problems suffice for optimal solution to the whole problem"

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 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.



step 1 – a recursive algorithm

- Suppose that like ordinary interval scheduling we have first sorted the requests by finish time f_i so

  $f_1 \leq f_2 \leq \cdots \leq f_n$
- Say request **i** comes **before** request **j** if **i< j**
- For any request **j** let **p(j)** be
  - the largest-numbered request before **j** that is compatible with **j**
  - or **0** if no such request exists
- Therefore **{1,…,p(j)}** is precisely the set of requests before **j** that are compatible with **j**

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,…,p(n)}**

  Not only that!

  Any set of requests in **{1,…,p(n)}** will be compatible with request **n**

  So in this case the optimal solution $\mathcal{O}$ must contain an optimal solution for **{1,…,p(n)}**

  "Principle of Optimality"

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 not** include request n then all requests in $\mathcal{O}$ must be contained in {1,…, n-1}
  - Not only that!
    - The optimal solution $\mathcal{O}$ must contain an optimal solution for {1,…, n-1}
    - **"Principle of Optimality"**

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:

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
  $$\mathbf{OPT}(n) = \max[w_n + \mathbf{OPT}(p(n)), \mathbf{OPT}(n-1)]$$
- Also
  $n \in \mathcal{O}$ iff $w_n + \mathrm{OPT}(p(n)) > \mathrm{OPT}(n-1)$

step 1 – a recursive algorithm

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

```
ComputeOpt(n)
    if n=0 then return(0)
    else
        u←ComputeOpt(p[n])
        v←ComputeOpt(n-1)
        if $w_n$+u>v then
            return($w_n$+u)
        else
            return(v)
    endif
```

step 2 – memoization

- 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 ComputeOpt
- Store these answers in an array OPT[n] and only recompute when necessary
  - Memoization
- Initialize OPT[i]=0 for i=1,…,n

step 2 – memoization

```
ComputeOpt(n):
    if n=0 then return(0)
    else
        u←MComputeOpt(p[n])
        v←MComputeOpt(n-1)
        if $w_n$+u>v then
            return($w_n$+u)
        else return(v)
    endif
```

```
MComputeOpt(n):
    if OPT[n]=0 then
        v←ComputeOpt(n)
        OPT[n]←v
        return(v)
    else
        return(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]←0
   for i=1 to n
        if w_i+OPT[p[i]] >OPT[i-1] then
            OPT[i] ←w_i+OPT[p[i]]
        else
            OPT[i] ←OPT[i-1]
        endif
   endfor
```

producing an optimal solution

```
IterativeComputeOptSolution(n)
  array OPT[0,…,n], Used[1,…,n]
  OPT[0]←0
  for i=1 to n
    if w_i+OPT[p[i]] >OPT[i-1] then
      OPT[i] ←w_i+OPT[p[i]]
      Used[i]←1
    else
      OPT[i] ← OPT[i-1]
      Used[i] ←0
    endif
  endfor
```

producing an optimal solution

```
IterativeComputeOptSolution(n)
  array OPT[0,…,n], Used[1,…,n]
  OPT[0]←0
  for i=1 to n
    if w_i+OPT[p[i]] >OPT[i-1] then
      OPT[i] ←w_i+OPT[p[i]]
      Used[i]←1
    else
      OPT[i] ← OPT[i-1]
      Used[i] ←0
    endif
  endfor
```

```
i ←n
S ← ∅
while i > 0 do
    if Used[i]=1 then
        S ← S ∪ {i}
        i ← p[i]
    else
        i ← i - 1
    endif
endwhile
```

example

	1	2	3	4	5	6	7	8	9
s_i	4	2	6	8	11	15	11	12	18
f_i	7	9	10	13	14	17	18	19	20
w_i	3	7	4	5	3	2	7	7	2
p[i]									
OPT[i]									
Used[i]									

example

	1	2	3	4	5	6	7	8	9
s_i	4	2	6	8	11	15	11	12	18
f_i	7	9	10	13	14	17	18	19	20
w_i	3	7	4	5	3	2	7	7	2
p[i]	0	0	0	1	3	5	3	3	7
OPT[i]									
Used[i]									

example

	1	2	3	4	5	6	7	8	9
s_i	4	2	6	8	11	15	11	12	18
f_i	7	9	10	13	14	17	18	19	20
w_i	3	7	4	5	3	2	7	7	2
p[i]	0	0	0	1	3	5	3	3	7
OPT[i]	3	7	7	8	10	12	14	14	16
Used[i]	1	1	0	1	1	1	1	0	1

example

	1	2	3	4	5	6	7	8	9
s_i	4	2	6	8	11	15	11	12	18
f_i	7	9	10	13	14	17	18	19	20
w_i	3	7	4	5	3	2	7	7	2
p[i]	0	0	0	1	3	5	3	3	7
OPT[i]	3	7	7	8	10	12	14	14	16
Used[i]	1	1	0	1	1	1	1	0	1

S={9,7,2}

segmented least squares

Least Squares

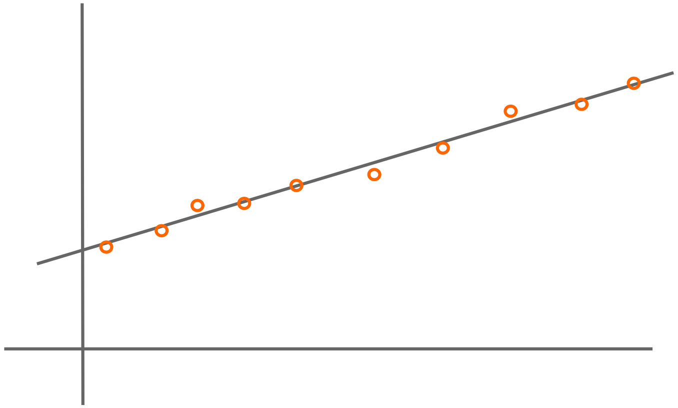
- **Given a set P of n points in the plane $p_1=(x_1,y_1),\ldots,p_n=(x_n,y_n)$ with $x_1<\ldots< x_n$ determine a line L given by $y=ax+b$ that optimizes the totaled 'squared error'**

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

- **A classic problem in statistics**
- **Optimal solution is known (see text)**

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

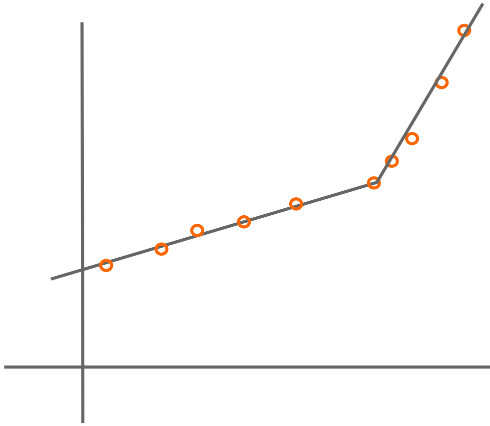
least squares



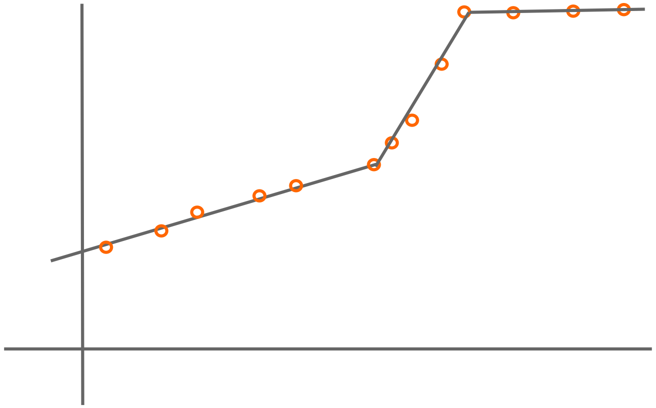
segmented least squares

What if data seems to follow a piece-wise linear model?

segmented least squares



segmented least squares



segmented least squares

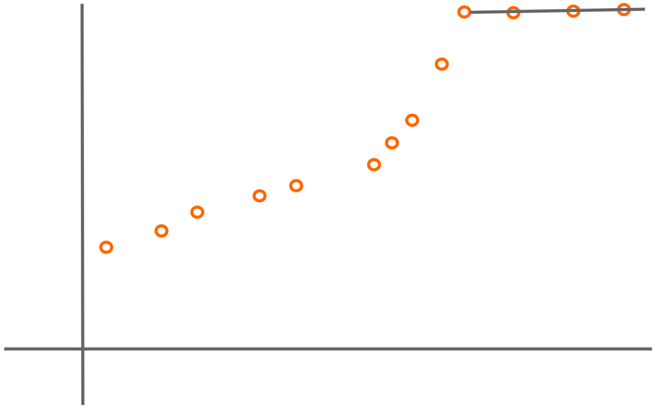
- What if data seems to follow a piece-wise linear model?
- Number of pieces to choose is not obvious
- If we chose **n-1** pieces we could fit with **0** error
  - Not fair
- Add a penalty of **C** times the number of pieces to the error to get a total penalty
- How do we compute a solution with the smallest possible total penalty?

segmented least squares

Recursive idea

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

segmented Least Squares



segmented least squares

Recursive idea

- We don't know which point is p_j
  - But we do know that $1 \le j \le n$
  - The optimal choice will simply be the best among these possibilities
- Therefore:

$$\mathrm{OPT}(\boldsymbol{n}) = \min_{1\le j\le n}\{\mathrm{Error}(\{\boldsymbol{p_j},\dots,\boldsymbol{p_n}\}) + \boldsymbol{C} + \mathrm{OPT}(\boldsymbol{j}-\mathbf{1})\}$$

dynamic programming solution

```
SegmentedLeastSquares(n)
   array OPT[0,...,n], Begin[1,...,n]
   OPT[0]←0
   for i=1 to n
    OPT[i]←Error{(p_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])
```