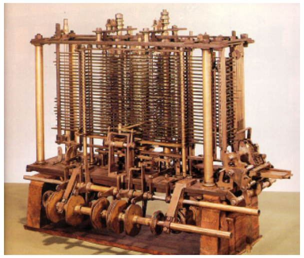
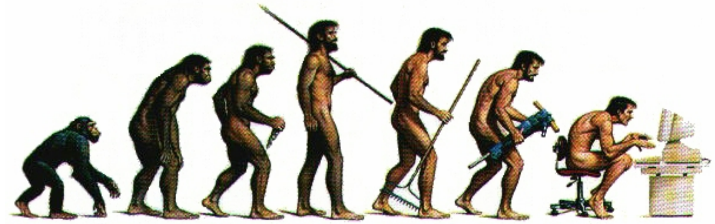




Data Structures and Algorithms II

CMPSC 130B

Announcements (Feb 3)

- Homework 2 in two parts
  - Problems except #4, #7 due today
  - #4, #7 due Friday
- Midterm Monday Feb 8
  - Scheduling details
    - 2 hour to solve + 15 minutes to upload on Gradescope
    - 9-11:15 on Monday
    - Posted on Piazza at 9
    - Clarifications on Zoom or Piazza 9-11
  - Topics up to today's lecture
- Today's plan
  - Dynamic programming

Dynamic Programming (DP)

- A powerful paradigm for algorithm design.
- Often leads to elegant and efficient algorithms when greedy or divide-and-conquer don't work.
- DP also breaks a problem into subproblems, but subproblems are not independent.
  - Tabulates solutions of subproblems to avoid solving them again (memoization).
- Typically applied to optimization problems: many feasible solutions; find one of optimal value.
- Key is the principle of optimality: solution composed of optimal subproblem solutions.

DP Applications

- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, compilers, systems, ….
- Some famous dynamic programming algorithms.
  - Unix diff for comparing two files.
  - Viterbi for hidden Markov models.
  - Smith-Waterman for genetic sequence alignment.
  - Bellman-Ford for shortest path routing in networks.
  - Cocke-Kasami-Younger (CKY) for parsing context free grammars.

Remember Fibonacci numbers

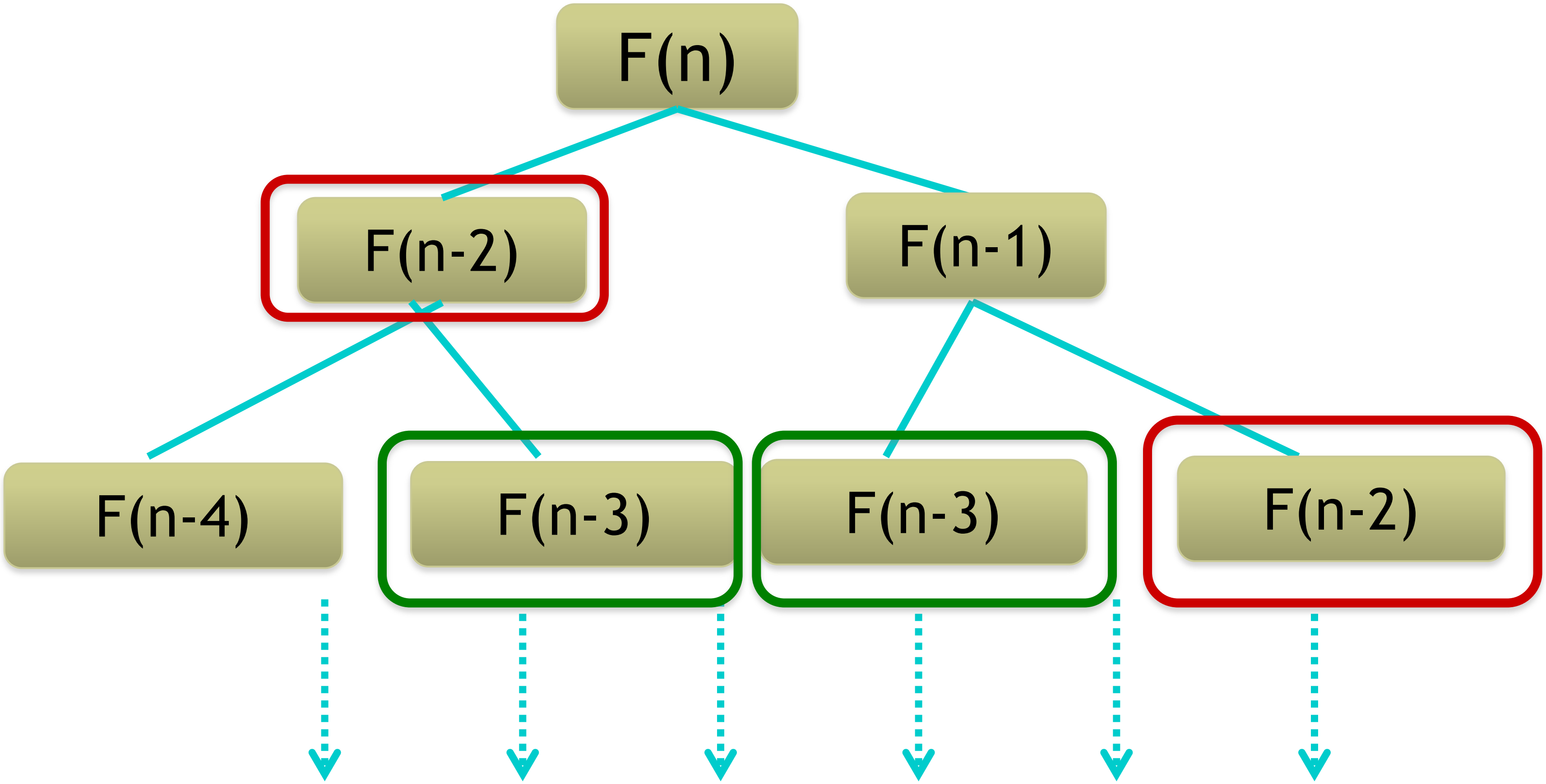
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

$$F(0) = 0, F(1)=1$$
$$F(n) = F(n-1) + F(n-2)$$

If we are not careful, a recursive algorithm ends up with an exponential number of redundant recursive calls.

However, not many of these are distinct.

How many distinct recursive calls?



Answer: only n because we only have F(i), i=1 … n.

Store solutions to subproblems each time they're solved (memoization).

DP Origin (Richard Bellman)

Richard Bellman: An interesting question is, ‘Where did the name, dynamic programming, come from?’ The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I’m not using the term lightly; I’m using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research. You can imagine how he felt, then, about the term, mathematical. … Hence, I felt I had to do something to shield Wilson … from the fact that I was really doing mathematical research inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning…. But planning, is not a good word for various reasons. I decided therefore to use the word, ‘programming.’ … Then, I said let’s take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it’s impossible to use the word, dynamic, in a pejorative/negative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It’s impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it to hide my activities.

What does it do?

- It often improves an otherwise exponential time (exhaustive search) algorithm to a polynomial time algorithm.
- This is a very powerful method, you really need to master it very well.
- Let us revisit the coin change problem.

DP Examples

- Coin change
- Weighted Interval Scheduling
- Least squares
- Matrix multiplication
- Longest Common Subsequence
- Sequence alignment
- Subset sum
- Knapsack
- Shortest paths

DP: Optimal coin change

- We have seen this problem before: you are given an amount in cents, and you want to make change with the smallest number of coins possible.
- Sometimes a greedy algorithm gives the optimal solution.
- But sometimes it does not: For example, for the coin system (12, 5, 1), the greedy algorithm gives 15 = 12 + 1 + 1 + 1 but a better answer is 15 = 5 + 5 + 5.
- Sometimes Greedy cannot even find the proper change: change for 41 cents using (25,10,4).
- So how can we always find the optimal solution (fewest coins)? One way is using dynamic programming.

Optimal coin change

- The idea: go bottom up.
  - To make change for n cents, the optimal method must use some denomination d_i.
  - That is, the optimal chooses the optimal solution for $n - d_i$ for some d_i and adds one coin of d_i to it.
  - We don't know which d_i to use, but some must work.
  - So, we try them all, assuming we know how to make optimal changes for < n cents. (Principle of Optimality)
- Let $OPT[j]$ be the optimal number of coins to make change for j cents, we have:

$$OPT[0] = 0$$

$$\mathrm{OPT}[j] = 1 + \min_{d_i \le j} OPT[j - d_i]$$

Example

- Suppose we use the system of denominations (1,5,18,25). To represent 29, the greedy algorithm gives 25+1+1+1+1 with 5 coins.
- DP algorithm: best coin 18 or 5 or 1. Consider 18, 29-18 = 11.
- 11 has a representation of size 3, with best coin 5. 11-5=6.
- 6 has a representation of size 2, with best coin 5. 6-5=1.
- So DP gives 29 = 18 + 5 + 5 + 1, with 4 coins.

j	opt[j]	best[j]
1	1	1
2	2	1
3	3	1
4	4	1
5	1	5
6	2	5
7	3	5
8	4	5
9	5	5
10	2	5
11	3	5
12	4	5
13	5	5
14	6	5

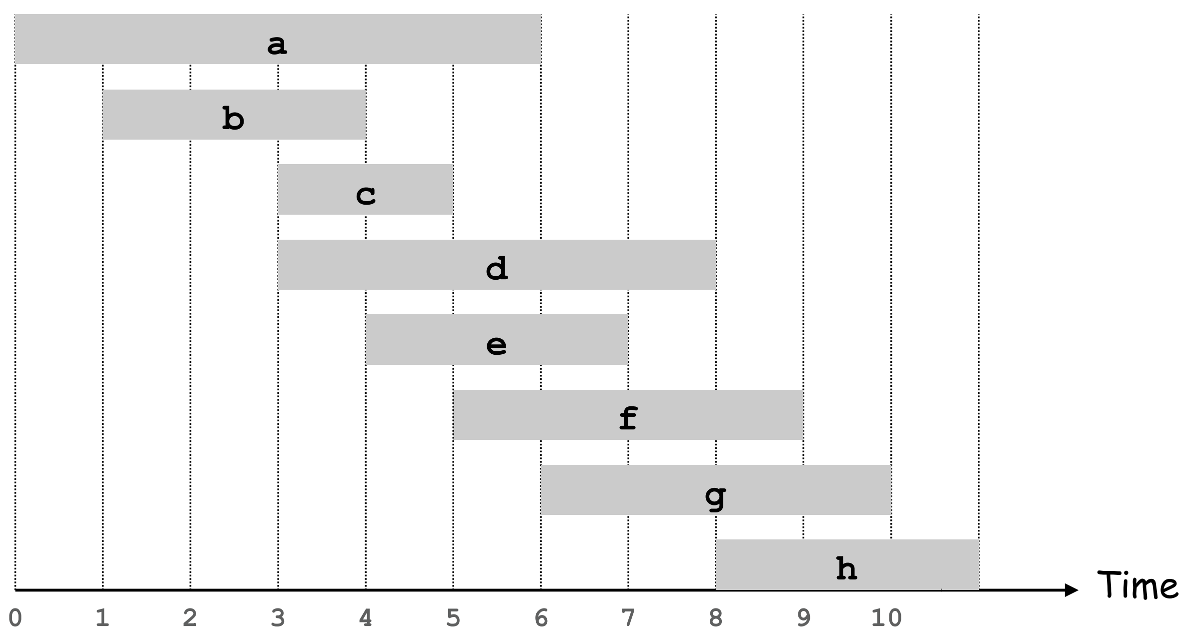
j	opt[j]	best[j]
15	3	5
16	4	5
17	5	5
18	1	18
19	2	18
20	3	18
21	4	18
22	5	18
23	2	18
24	3	18
25	1	25
26	2	25
27	3	25
28	3	18
29	4	18

Notes

What happens to the DP recurrence if Greedy gives optimal result?

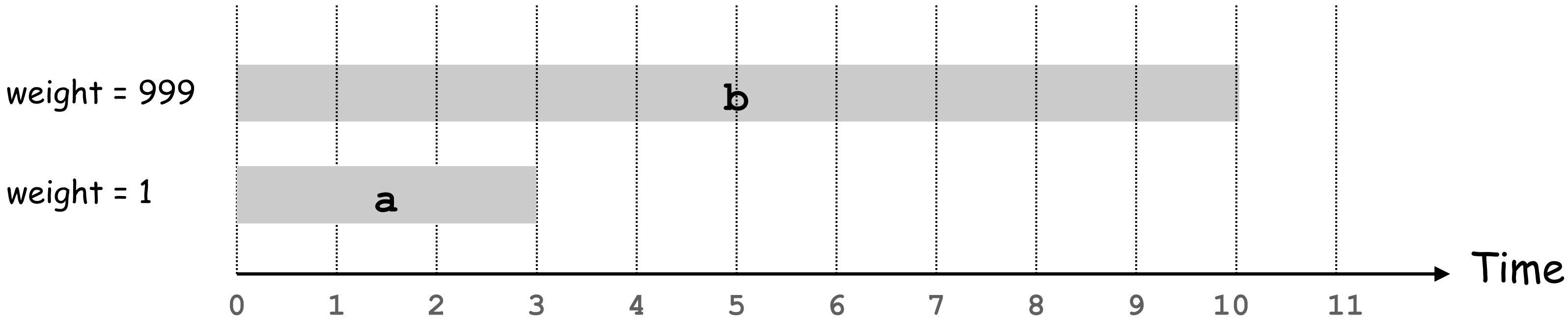
Weighted Interval Scheduling

- Job j starts at s_j, finishes at f_j, and has weight/value v_j.
- Two jobs are compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.



Does Greedy work in this case?

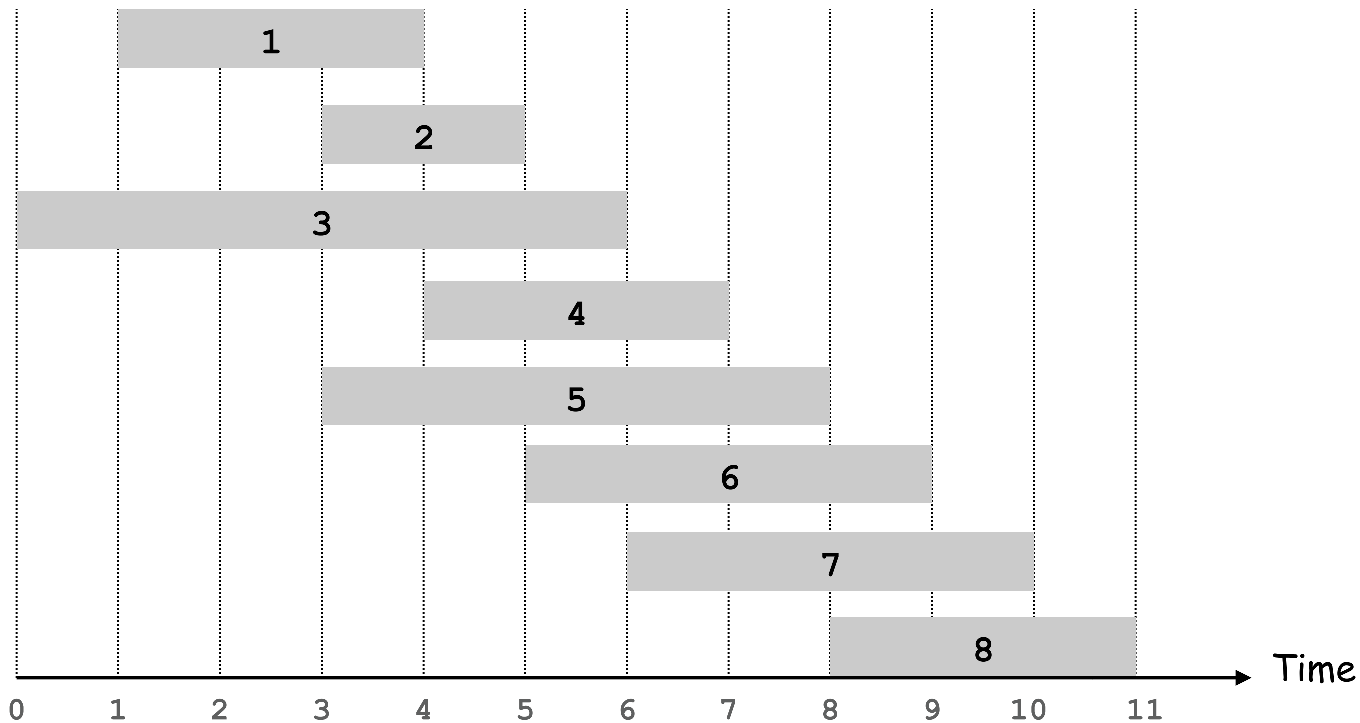
- Unfortunately, no
- Need to apply DP



Weighted Interval Scheduling

- Label jobs by finishing time: $f_1 \leq f_2 \leq . . . \leq f_n$.
- Define p(j) = largest index i < j such that job i is compatible with j.

  p(8) = 5, p(7) = 3, p(2) = 0.



Choosing the optimal subproblem

Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- Case 1: OPT selects job j.
  - collect profit v_j
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

- Case 2: OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0, & if\ j = 0 \\ \max\left(v_j + OPT\left(p(j)\right), OPT(j-1)\right), & otherwise \end{cases}$$

Memoization

- Store results of each sub-problem in a table; lookup as needed.
- Can store information on the optimal choice at each step
- Recursive or iterative solutions possible.

Time complexity

- Sort by finish time
- Build array p
- Fill out OPT
- $O(n \log n)$, where n is the number of jobs

Notes

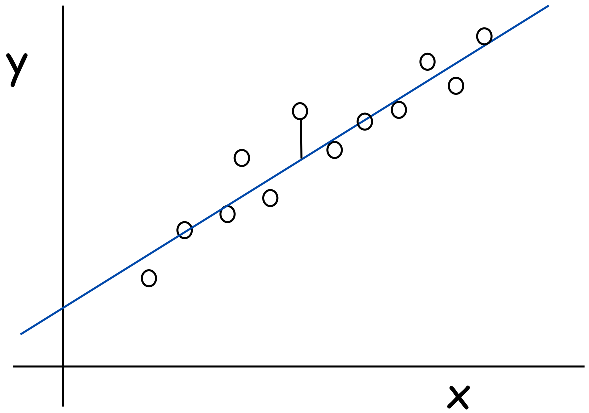
Breakout: Can we set up the DP recurrence by sorting the jobs by start time (instead of finish time) and looking for compatibility based on finish time? How?

Least squares problem

- A fundamental problem in machine learning
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- Find a line $y = ax + b$ that minimizes the sum of the squared error:

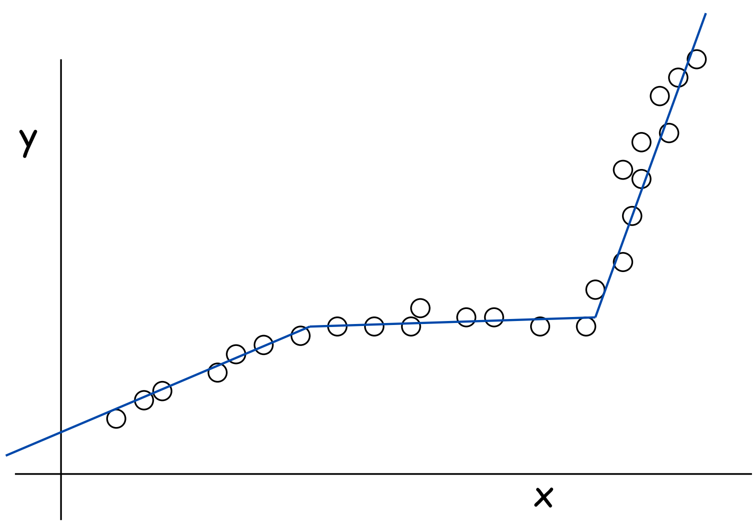
$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

- Solve by differentiating



Segmented least squares

- Points lie roughly on a sequence of several line segments
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that maximizes accuracy and parsimony
- Measure Accuracy: E = sum of SSE in each segment
- Measure parsimony: the number of lines L
- Tradeoff function: $E + c\,L$, for some constant $c > 0$



Choosing the optimal subproblem

- OPT(j) = minimum cost for points $p_1, ..p_i , . . . , p_j$.
- e(i, j) = SSE for points $p_i, p_{i+1} , . . . , p_j$ using best segment.
- To compute OPT(j):
  - Last segment uses points $p_i, p_{i+1} , . . . , p_j$
  - Cost = e(i, j) + c + OPT(i-1).

$$OPT(j) = \begin{cases} 0 & , \quad if\ j = 0 \\ \min_{1 \le i \le j} \{e(i,j) + c + OPT(i-1)\}, & otherwise \end{cases}$$

- Running time ?

Notes

Announcements (Feb 10)

- Mid-quarter survey on GauchoSpace
- Feedback on midterm
- Future assignments (note changes)
  - PA2 handed out Wed Feb 10, Due Friday March 12
  - HW3 handed out Wed Feb 17, Due Friday Feb 26
  - HW4 handed out Monday March 1, Due Monday March 8
- Today’s plan: dynamic programming
  - Matrix multiplication
  - Longest Common Subsequence
  - String alignment

Matrix Multiplication: Review

- Suppose that M_1 is of size p_0 x p_1, and M_2 is of size p_1 x p_2.
- What is the time complexity of computing $M_1 \times M_2$?
- What is the size of the result?

Matrix Multiplication: Review

- Suppose that M_1 is of size p_0 x p_1, and M_2 is of size p_1 x p_2.
- What is the time complexity of computing $M_1 \times M_2$?
- What is the size of the result? p_0 x p_2.
- Each number in the result is computed in $O(p_1)$ time by:
  - multiplying p_1 pairs of numbers.
  - adding p_1 numbers.
- Overall time complexity: $O(p_0 \text{ x } p_1 \text{ x } p_2)$.

Optimal Ordering

- Suppose that we need to do a sequence of matrix multiplications:

  Result = $M_1 \times M_2 \times M_3 \times \ \ldots \times M_n$

- Number of columns for M_i = number of rows for $M_{i+1} = p_i$
  - Dimension of M_1 =
  - Dimension of M_2 =
- What is the time complexity for performing this sequence of multiplications?

Optimal Ordering

- Suppose that we need to do a sequence of matrix multiplications:

  Result = $M_1 \times M_2 \times M_3 \times \ \ldots \times M_n$

- Number of columns for M_i = number of rows for $M_{i+1} = p_i$
  - Dimension of M_1 =
  - Dimension of M_2 =
- What is the time complexity for performing this sequence of multiplications?
  - Depends on the order.

An Example

- Suppose:
  - M_1 is 17x2.
  - M_2 is 2x35.
  - M_3 is 35x4.
- $(M_1 \times M_2) \times M_3$:
- $M_1 \times (M_2 \times M_3)$:

An Example

- Suppose:
  - M_1 is17x2.
  - M_2 is 2x35.
  - M_3 is 35x4.
- $(M_1 \times M_2) \times M_3$:
  - 17*2*35 = 1190 multiplications and additions to compute $M_1 \times M_2$.
  - 17*35*4 = 2380 multiplications and additions to compute multiplying the result of $(M_1 \times M_2)$ with M_3.
  - Total: 3570 multiplications and additions.
- $M_1 \times (M_2 \times M_3)$:
  - 2*35*4 = 280 multiplications and additions to compute $M_2 \times M_3$.
  - 17*2*4 = 136 multiplications and additions to compute multiplying M_1 with the result of $(M_2 \times M_3)$.
  - Total: 416 multiplications and additions.

Dynamic Programming Approach

Suppose that we need to do a sequence of matrix multiplications:

Result = $M_1 \times M_2 \times M_3 \times \ \ldots \times M_n$

For dynamic programming, we must address two questions:

1. Can we define a set of smaller problems, such that the solutions to those problems make it easy to solve the original problem?
2. Can we arrange those smaller problems in a sequence so that each problem in that sequence only depends on problems that come earlier in the sequence?

Matrix Chain

- Consider 4 matrices: M_1, M_2, M_3, M_4.
- We can compute the product in many ways, depending on how we parenthesize.

$$M_1 (M_2 (M_3 M_4))$$

$$M_1 ((M_2 M_3) M_4)$$

$$(M_1 M_2) (M_3 M_4)$$

$$(M_1 (M_2 M_3)) M_4$$

$$((M_1 M_2) M_3) M_4$$

- We need to find the optimal order of placing parentheses
  - Note that matrix multiplication is associative.

How many choices?

- Given a chain of $n \geq 2$ matrices to multiply, we need to compute the number of ways to fully parenthesize. Call this value P(n).

$$P(n) = \begin{cases} 1 & , \quad n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) , & n \geq 2 \end{cases}$$

k defines the last matrix product

- Let us compute some values of P(n)
- Defines Catalan numbers
- Grows as $\frac{4^n}{n^{\frac{3}{2}}}$ or $\binom{2n}{n}$

DP Recurrence

- A subproblem is a subchain M_i , M_{i+1} . . . , M_j
- m[i, j] = optimal cost for this chain
- Use principle of optimality to determine m[i, j] recursively
- Clearly, m[i, i] = 0, for all i
- If an algorithm computes M_i , M_{i+1} . . . , M_j as

  $(M_i , . . . , M_k) \times (M_{k+1}, . . . , M_j)$, then

  $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$

- $m[i,j] = \min_{i \le k < j} \{ m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j \}$

The DP Approach

- Naive recursion is exponential because it solves the same subproblem over and over in different branches of recursion
- DP avoids this wasted computation by organizing the subproblems differently: bottom up based on length
- Start with $m[i, i] = 0$, for all i
- Next, we determine $m[i, i + 1]$, and then $m[i, i + 2]$, and so on

The Algorithm

- Input: $[p_0, p_1, \ldots, p_n]$ the dimension vector of the matrix chain
- Output: $m[i, j]$, the optimal cost of multiplying each subchain $M_i \times \ldots \times M_j$.
- Array $s[i, j]$ stores the optimal k for each subchain

The Algorithm

1. Set m[i, i] = 0, for i = 1, 2, . . . , n
2. Set d = 1
3. For all i, j such that j − i = d,

   $$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\}$$

   Set s[i, j] = k*, the choice that gives min value in above expression.

4. Increment d and repeat Step 3

Example

$M1 = 30 \times 35$, $M_2 = 35 \times 15$, $M_3 = 15 \times 5$, $M_4 = 5 \times 10$, $M_5 = 10 \times 20$, $M_6 = 20 \times 25$

j

i

	1	**2**	**3**	**4**	**5**	**6**
1	0	15,750	7,875	9,375	11,875	15,125
2		0	2,625	4,375	7,125	10,500
3			0	750	2,500	5,375
4				0	1,000	3,500
5					0	5,000
6						0

Running time = $O(n^3)$

Quiz: Show the s matrix

Notes

Longest Common Subsequence (LCS)

- Given two sequences X and Y, find their longest common subsequence.
- If X = (A, B, C, B, D, A, B) and Y = (B, D, C, A, B, A), then (B, C, A) is a common subsequence, but not LCS.
- (B, D, A, B) is an LCS.
- How do we find an LCS?

Facts about LCS

- Suppose $Z = (z_1, z_2, \dots, z_k)$ is an LCS of X[1..m] and Y[1..n].
- Then,

  If $x_m = y_n$ then

  $z_k = x_m = y_n$ and

  Z[1..k-1] = LCS(X[1..m-1], Y[1..n-1]).

  Otherwise

  $z_k \neq x_m$ implies Z = LCS(X[1..m-1], Y)

  $z_k \neq y_n$ implies Z = LCS(X, Y[1..n-1])

Recurrence

- Let $c[i,j] = |LCS(X[1..i], Y[1..j])|$ be the optimal solution for $X[1..i], Y[1..j]$.
- Then,

$$c(i,j) = \begin{cases} 0, & if\ i = 0\ or\ j = 0 \\ c(i-1,j-1)+1, & if \quad x_i = y_j \\ \max\{c(i,j-1), c(i-1,j)\}, & if \quad x_i \neq y_j \end{cases}$$

Example

i

j

		A	**B**	**C**	**D**	**A**	**B**
		0	0	0	0	0	0
B	0	0	1	1	1	1	1
D	0	0	1	1	2	2	2
C	0	0	1	2	2	2	2
A	0	0	1	2	2	3	3
B	0	0	1	2	2	3	4
A	0	0	1	2	2	3	4

LCS = BDAB

Running time = O(mn)

Notes

Announcements (Feb 17)

- Mid-quarter survey on GauchoSpace
- Current/future assignments
  - PA2 handed out Wed Feb 10, Due Friday March 12
  - HW3 handed out today, Due Friday Feb 26
  - HW4 handed out Monday March 1, Due Monday March 8
- Today's plan: dynamic programming
  - Longest Common Subsequence
  - Setting up optimizations
  - Optimal Binary Search Tree
  - String alignment

How does LCS recurrence extend to k strings?

- Let $c[i,j] = |LCS(X[1..i], Y[1..j])|$ be the optimal solution for $X[1..i], Y[1..j]$.
- Then,

$$c(i,j) = \begin{cases} 0, & if\ i = 0\ or\ j = 0 \\ c(i-1, j-1) + 1, & if \quad x_i = y_j \\ \max\{c(i, j-1), c(i-1, j)\}, & if \quad x_i \neq y_j \end{cases}$$

Substrings without gaps

- Let $c[i,j] = |LCSNG(X[1..i], Y[1..j])|$ be the optimal solution for $X[1..i], Y[1..j]$, assuming that the common strings end at i and j respectively.
- Then,

$$c(i,j) = \begin{cases} 0, & if\ i = 0\ or\ j = 0 \\ c(i-1, j-1) + 1, & if \quad x_i = y_j \\ 0, & if \quad x_i \neq y_j \end{cases}$$

- Where to find the optimal solution?

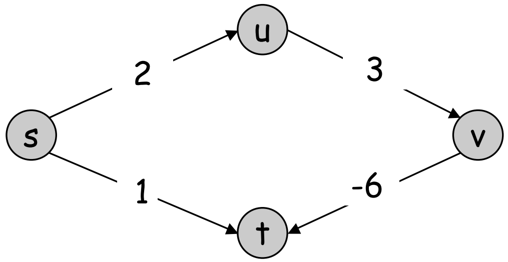
DP for other problems

- Huffman coding?
- Minimum spanning tree?

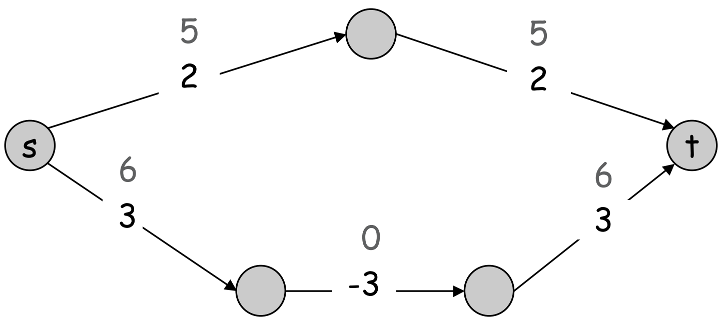
Shortest Paths

Dijkstra. Can fail if negative edge costs.

Which path will Dijkstra find?



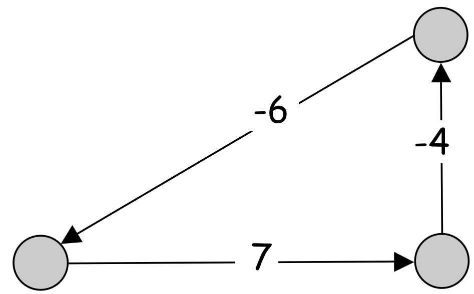
Re-weighting. Adding a constant to every edge weight can fail.



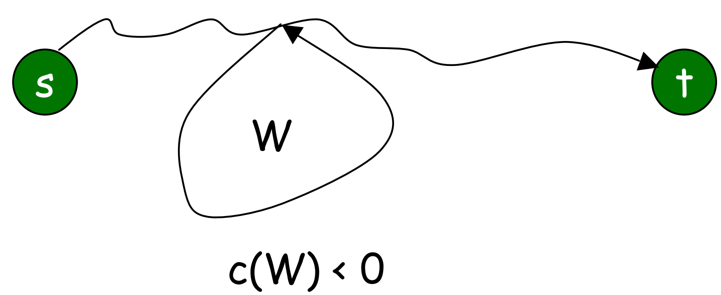
Which path will Dijkstra find?

Shortest Paths: Negative Cycles

Negative cost cycle.



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.



Shortest Paths: Dynamic Programming

Def. OPT(i, v) = length of shortest v-t path P using at most i edges.

- Case 1: P uses at most i-1 edges.
  - OPT(i, v) = OPT(i-1, v)

Bellman-Ford

- Case 2: P uses exactly i edges.
  - if (v, w) is first edge, then OPT uses (v, w), and then selects best w-t path using at most i-1 edges

$$OPT(i, v) = \begin{cases} 0 & \text{if } i = 0 \\ \min\left\{ OPT(i-1,\ v)\ ,\ \min\limits_{(v,w) \in E} \left\{ OPT(i-1,\ w) + c_{vw} \right\} \right\} & \text{otherwise} \end{cases}$$

Remark. By previous observation, if no negative cycles, then OPT(n-1, v) = length of shortest v-t path.

Distributed algorithm

Independence of subproblems

- Unweighted shortest (simple) paths versus unweighted longest (simple) paths
- Possible to decompose a shortest path p from u to v into shortest path p1 from u to w and shortest path p2 from w to v.
  – Can find shortest path p by considering all intermediate vertices w
- What about the decomposition of longest paths?
  – Do we have independence?

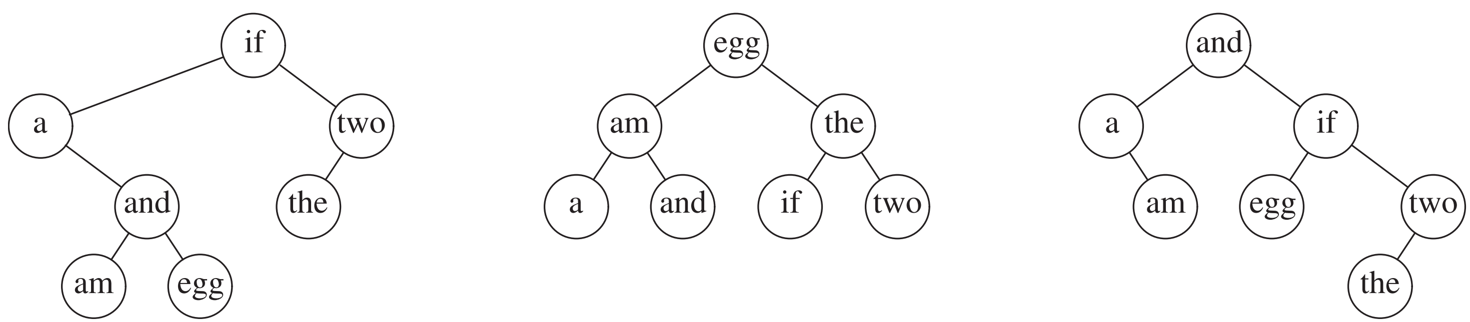
Announcements (Feb 22)

- Current/future assignments
  - PA2 due Friday March 12
  - HW3 due Friday Feb 26
  - HW4 handed out Monday March 1, Due Monday March 8
- Today's plan:
  - DP
    - Optimal Binary Search Tree
    - String alignment
    - Subset sum
    - Knapsack
  - Computational intractability (Classes NP and P)

Optimal BST

- Word w_i accessed with probability p_i
- Suppose w_i occurs at depth d_i
  - We want to minimize $\sum_{i=1}^{n} p_i(d_i + 1)$

Word	Probability
a	0.22
am	0.18
and	0.20
egg	0.05
if	0.25
the	0.02
two	0.08



greedy balanced optimal

How is this different from Huffman tree?

Optimal BST Recurrence

- OPT(i,j) = optimal BST cost for words $w_i,..,w_j$
  - Assume sorted order
  - We wish to find OPT(1,n)

$$OPT(i,i) = p_i$$
$$OPT(i,j) = 0, if\ i > j$$

$$OPT(i,j) = \min_{i \le r \le j} \left\{ OPT(i, r-1) + \sum_{k=i}^{r-1} p_k + p_r + OPT(r+1, j) \ + \sum_{k=i+1}^{j} p_k \right\}$$

$$= \min_{i \le r \le j} \left\{ OPT(i, r-1) + OPT(r+1, j) \ + \sum_{k=i}^{j} p_k \right\}$$

Time complexity = $O(n^3)$

Notes

Sequence Alignment

- Sequence is the most prevalent data
  - NLP text / string analysis
  - Biology
- Sequence alignment has multiple uses in biology
  - Detecting orthologs
  - Predicting function
  - Understanding evolution of coronavirus SARS-COV2
- Global and local alignment

A simple alignment

- Let us try to align two short nucleotide sequences:
  - AATCTATA and AAGATA
- Without considering any gaps (insertions/deletions) there are 3 possible ways to align these sequences
- Which one is best?

```
AATCTATA      AATCTATA      AATCTATA
AAGATA         AAGATA         AAGATA
```

Scoring alignments

- We need to have a scoring mechanism to evaluate alignments
  - match score
  - mismatch score
- We can have the total score as:

$$\sum_{i=1}^{n} \text{match or mismatch score at position i}$$

- For the simple example, assume a match score of 1 and a mismatch score of 0:

```
AATCTATA        AATCTATA        AATCTATA
AAGATA           AAGATA            AAGATA
```

4 1 3

Need to allow gaps

- Maximal consecutive run of spaces in alignment
  - Matching mRNA (cDNA) to DNA
  - Shortening of DNA during replication
  - Unequal cross-over during meiosis
  - …
- Need a scoring function that considers gaps

Alignment with gaps

- Considering gapped alignments vastly increases the number of possible alignments:

AATCTATA	**AATCTATA**	**AATCTATA**	more?
AAG-AT-A	**AA-G-ATA**	**AA--GATA**	
1	3	3	

f(m,n) = number of alignments of two strings of length m,n

$= f(m,n-1) + f(m-1,n) + f(m-1,n-1)$

- If gap penalty is -1 what will be the new scores?

More complicated gap penalties

- Nature favors a small number of long gaps compared to a large number of short gaps.
- How do we adjust our scoring scheme to account for this fact above?

By having different gap opening and gap extension penalties.

- Choices of gap penalties
  - Linear
  - Affine
    - Gap open penalty
    - Gap extension penalty
  - Convex
  - Arbitrary

Global sequence alignment (Needleman-Wunsch)

- Think distance instead of score
- Edit distance, $d(a,b)$ = distance between symbols a and b
  - Linear gap model
- $D(i,j)$ = edit distance between strings $\alpha(1:i)$ and $\beta(1:j)$
- Recurrence relation
  - $D(i,0) = \sum d(\alpha(k),-),\ 1 \le k \le i$
  - $D(0,j) = \sum d(-, \beta(k)),\ 1 \le k \le j$
  - $D(i,j) = \min\ [D(i-1,j) + d(\alpha(i),-),$ ← i opposite gap
    $D(i,j-1) + d(-, \beta(j)),$ ← j opposite gap
    $D(i-1,j-1) + d(\alpha(i), \beta(j))]$ ← i opposite j

Example

α: G C T G G A A G G C A – T
β: G C – A G – A – G C A C G

Linear gap model
Match = 0
Mismatch = 1

α

β	--	G	C	T	G	G	A	A	G	G	C	A	T
--	0	1	2	3	4	5	6	7	8	9	10	11	12
G	1	0	1	2	3	4	5	6	7	8	9	10	11
C	2	1	0	1	2	3	4	5	6	7	8	9	10
A	3	2	1	1	2	3	3	4	5	6	7	8	9
G	4	3	2	2	1	2	3	4	4	5	6	7	8
A	5	4	3	3	2	2	2	3	4	5	6	6	7
G	6	5	4	4	3	2	3	3	3	4	5	6	7
C	7	6	5	5	4	3	3	4	4	4	4	5	6
A	8	7	6	6	5	4	3	3	4	5	5	4	5
C	9	8	7	7	6	5	4	4	4	5	5	5	5
G	10	9	8	8	7	6	5	5	4	4	5	6	6

Notes

Subset Sum Problem

- Let $w_1,\ldots,w_n = \{6, 8, 9, 11, 13, 16, 18, 24\}$
- Find a subset that has as large a sum as possible, without exceeding 50

Adding a parameter for weight

- OPT[j, K] = the highest weight subset of $\{w_1, \ldots, w_j\}$ with weight at most K
- {2, 4, 7, 10}
  - OPT[2, 7] =
  - OPT[3, 7] =
  - OPT[3,12] =
  - OPT[4,12] =

Subset Sum recurrence

- OPT[j, K] = highest weight subset of $\{w_1, \ldots, w_j\}$ with weight at most K

$$
\begin{aligned}
OPT[j, K] \\
&= \max(OPT[j-1, K], OPT[j-1, K-w_j] + w_j), \; if\ w_j \le k \\
&= OPT[j-1, K], otherwise
\end{aligned}
$$

Do we need to sort the values?

Subset Sum Matrix

{2, 4, 7, 10}

Compute OPT[4,12]

j																	
4	0																
3	0																
2	0																
1	0																
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

K

Breakout

Knapsack Problem

- Items have weights and values
- The goal is to maximize total value subject to a bound W on total weight
- Items $\{I_1, I_2, \ldots I_n\}$
  – Weights $\{w_1, w_2, \ldots, w_n\}$
  – Values $\{v_1, v_2, \ldots, v_n\}$

Example

Ex: { 3, 4 } has value 40.

`W = 11`

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Greedy: repeatedly add item with maximum ratio v_i / w_i.
Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

DP by adding a new variable

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w

- Case 2: OPT selects item i.
  - new weight limit = w – w_i
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } \mathrm{i} = 0 \\ OPT(i-1,w) & \text{if } \mathrm{w_i} > \mathrm{w} \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i)\,\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

```
Input: n, W, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

Example

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Time complexity

- Running time $\Theta(n\, W)$.
  - Not polynomial in input size!
  - "Pseudo-polynomial."
  - Decision version of Knapsack is NP-complete. [Chapter 8]
- Knapsack approximation algorithm. There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

Notes