CSE 417
Dynamic Programming (pt 2)
Look at the Last Element

UNIVERSITY *of* WASHINGTON

Reminders

- **HW4 is due on Friday**
  - start early!
  - if you run into problems loading data (date parsing), try running java with `-Duser.country=US -Duser.language=en`

Dynamic Programming Review

- Apply the steps...
  1. Describe solution in terms of solution to *any* sub-problems
  2. Determine all the sub-problems you'll need to apply this recursively
  3. Solve every sub-problem (once only) in an appropriate order

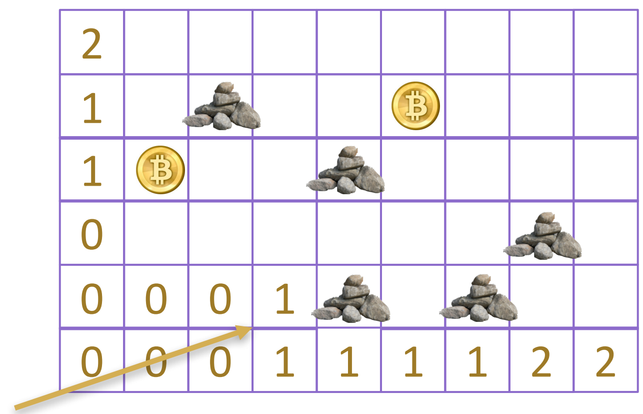
- Key question:
  1. Can you solve the problem by combining solutions from sub-problems?

- Count sub-problems to determine running time
  - total is number of sub-problems times time per sub-problem

Review From Last Time

> **Bitcoin Mining Broken Robot**

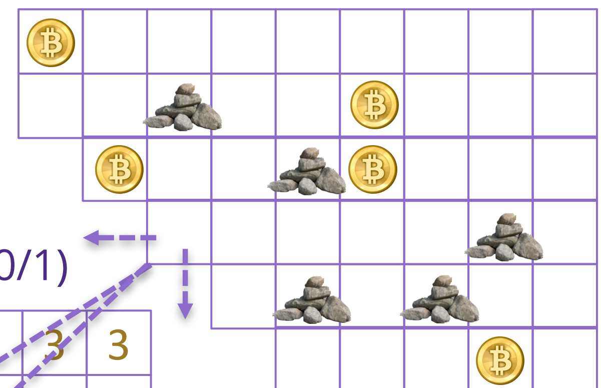
- sub-problems: where robot starts
- max coins he can collect at (i,j) =
  max(max coins he can collect at (i-1,j),
  max coins he can collect at (i,j-1))
  + 1 if coin at (i,j)
- solve from bottom-left to top-right



Review From Last Time

> **Bitcoin Mining Broken Robot**
>
> – sub-problems: where robot starts

> **Bitcoin Mining Bomber Robot**
>
> – sub-problems: where robot starts & if has bomb (0/1)
>
> – max of 4 options at (i,j,1):
>
>   > step left: (i-1,j,1)
>
>   > step down: (i,j-1,1)
>
>   > blast left: (i-1,j,0)
>
>   > blast right: (i,j-1,0)
>
>   – ignore rocks at that spot
>
> – still O(1) to calculate



2	2	2	2	2	3	3	3	3
1	2	0	2	2	3	3	3	3
1	2	2	2	0	2	2	2	2
0	0	0	1	1	1	1	0	2
0	0	0	1	0	1	0	2	2
0	0	0	1	1	1	1	2	2

Review From Last Time

> Can be implemented in Excel...

> Input Worksheet:

	A	B	C	D	E	F	G	H
1	coin							
2			rocks			coin		
3	coin	coin			rocks	coin		
4								rocks
5					rocks		rocks	
6	exit			coin				coin

Review From Last Time

> Can be implemented in Excel...

> No Bomb Worksheet:

fx =IF(Input!D5="rocks",-1,MAX(OptNoBomb!C5,OptNoBomb!D6)+IF(Input!D5="coin",1,0))

	A	B	C	D	E	F	G	H	I
1	2	2	2	2	2	3	3	3	3
2	1	2	-1	2	2	3	3	3	3
3	1	2	2	2	-1	2	2	2	2
4	0	0	0	1	1	1	1	-1	2
5	0	0	0	1	-1	1	-1	2	2
6	0	0	0	1	1	1	1	2	2

Review From Last Time

> Can be implemented in Excel...

> Blast Worksheet (ignore bombs at that spot):

```
=MAX(OptNoBomb!C5,OptNoBomb!D6)+IF(Input!D5="coin",1,0)
```

	A	B	C	D	E	F	G	H	I
1	2	2	2	2	2	3	3	3	3
2	1	2	2	2	2	3	3	3	3
3	1	2	2	2	2	2	2	2	2
4	0	0	0	1	1	1	1	2	2
5	0	0	0	1	1	1	1	2	2
6	0	0	0	1	1	1	1	2	2

Review From Last Time

> Can be implemented in Excel…

> With Bomb Worksheet:

fx =IF(Input!D5="rocks",-1,MAX(OptWithBomb!C5,OptWithBomb!D6,OptBlast!C5,OptBlast!D6)+IF(Input!D5="coin",1,0))

	A	B	C	D	E	F	G	H	I
1	2	2	2	2	2	4	4	4	4
2	1	2	-1	2	2	4	4	4	4
3	1	2	2	2	-1	3	3	3	3
4	0	0	0	1	1	1	1	-1	2
5	0	0	0	1	-1	1	-1	2	2
6	0	0	0	1	1	1	1	2	2

Outline for Today

- Weighted Interval Scheduling
- Max Sub-array Sum
- Document Layout in TeX
- Optimal Breakout Trades

Weighted Interval Scheduling

- **Problem**: Given a set of intervals $[s_1, e_1]$, ..., $[s_n, e_n]$ (start & end) with weights w_1, .., w_n, find the subset of *non-overlapping* intervals with *most total weight*.

- Would be strictly easier without weights
  - has a greedy algorithm (see textbook)
  - for similar reasons as to why Robot problem is easier with no coins
    - that one also has a greedy solution in that case (shortest path)
  - (will see something similar in the next topic:
    max flow is easier than min-cost flow)

Weighted Interval Scheduling

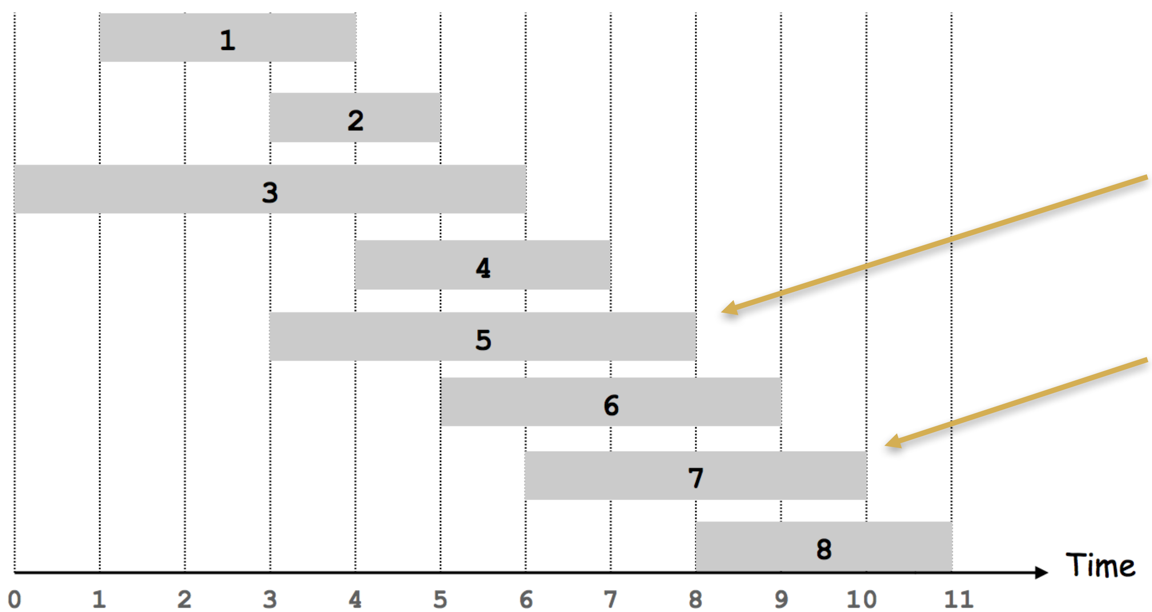
- Brute force: try all subsets...
  - 2^n subsets
  - for n = 300, this is larger than number of molecules in the universe

- Apply dynamic programming...
  - try to get from impossible to possible

Weighted Interval Scheduling

- Apply dynamic programming...
  - look for ways to solve the problem using the solution to sub-problems
- Q: What sub-problems would be useful?
- As with robot, often useful to think about the *last step* of solution
  - sub-problems told us how well we could do after a step left vs down
  - in this case, decisions are about whether to include each interval
  - consider: should we include the *last interval*?
    - what is the last one?
    - how about the one that finishes last

Weighted Interval Scheduling



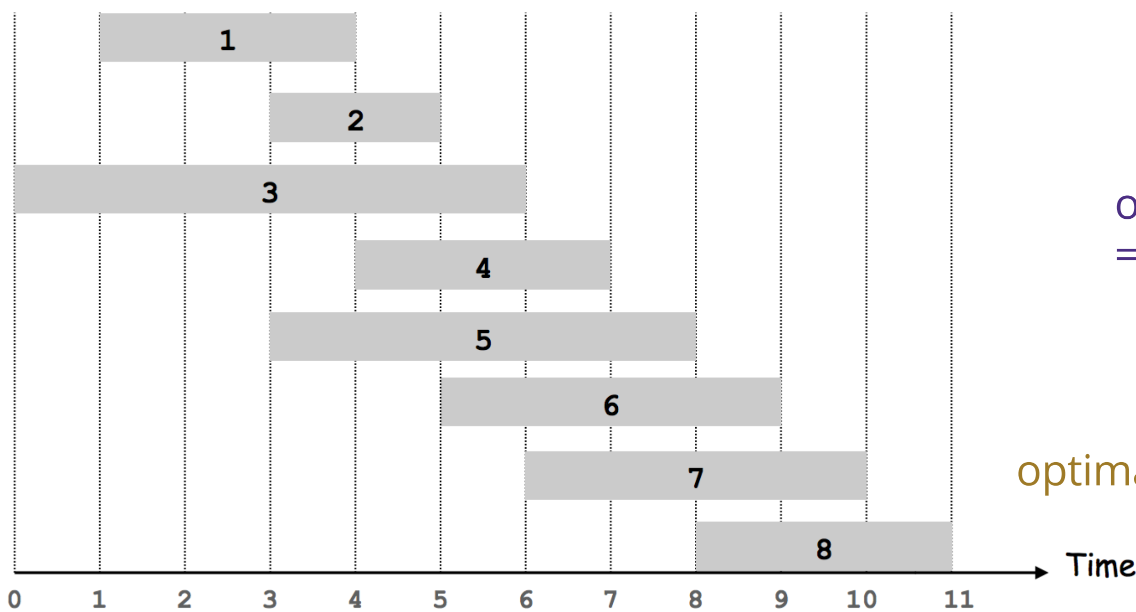
Two options:

1. include 8
2. don't include 8

if we include 8, can't use 6 or 7.
8 + any solution using [1, ..., 5]

if we don't include 8, then we can still have any solution over intervals [1, ..., 7]

Weighted Interval Scheduling



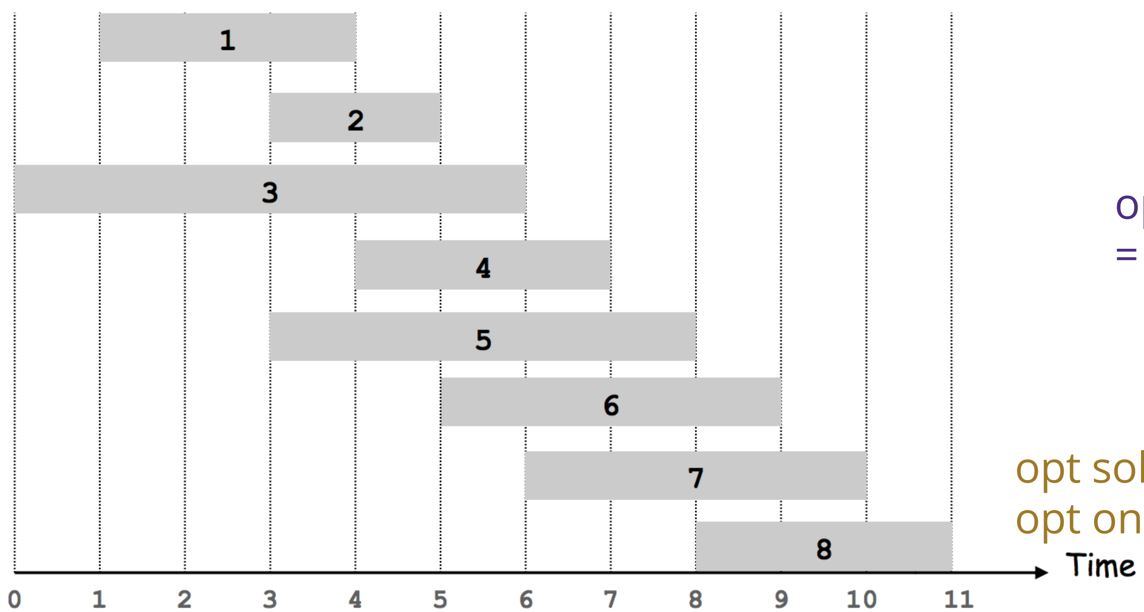
Two options:

1. include 8
2. don't include 8

opt value over [1, …, 8]
= max(opt value over [1, …, 7],
w_8 + opt value over [1, …, 5])

optimal substructure

Weighted Interval Scheduling



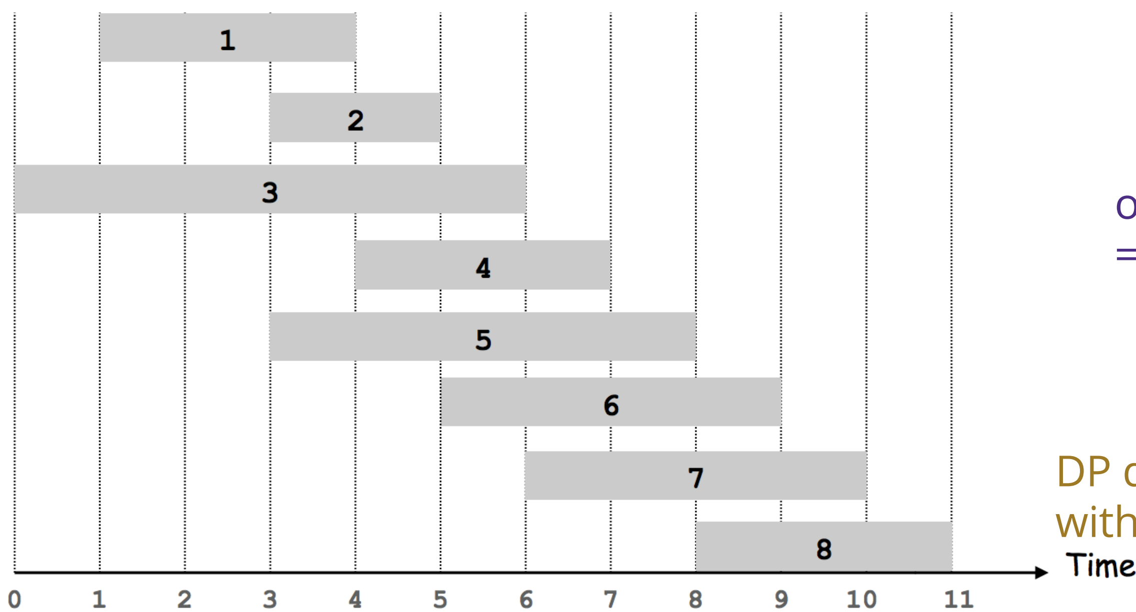
Two options:

1. include 8
2. don't include 8

opt value over [1, …, 8]
= max(opt value over [1, …, 7],
w_8 + opt value over [1, …, 5])

opt solution must be
opt on sub-problem

Weighted Interval Scheduling



Two options:

1. include 8
2. don't include 8

opt value over [1, …, 8]
= max(opt value over [1, …, 7],
w_8 + opt value over [1, …, 5])

DP does not work without this!

Weighted Interval Scheduling

- Order the elements by finish time
  - makes it easy to describe which ones can be used together

- Apply dynamic programming...
  1. Can find opt value for [1, ..., n] using only [1, ..., j] with $j < n$.
  2. Need solution to [1, ..., j] for each $j = 1, \dots, n$.
  3. Solve each of those starting with $j = 1$.
     - opt value for [1] = w_1
     - opt value for [1, ..., j] = max(
       opt value for [1, ..., j-1],
       w_j + opt value for [1, ..., i] where $e_i \le s_j$)

choose largest i for which this holds

Weighted Interval Scheduling

- Apply dynamic programming…
  1. Can find opt value for [1, …, n] using only [1, .., j] with $j < n$.
  2. Need solution to [1, …, j] for each $j = 1, \ldots, n$.
  3. Solve each of those starting with $j = 1$.
     - opt value for [1] = w_1
     - opt value for [1, …, j] = max(
       opt value for [1, …, j-1],
       w_j + opt value for [1, …, i] where $e_i \le s_j$)

- **Q**: How do we find the largest i with $e_i \le s_j$?
- **A**: binary search

Weighted Interval Scheduling

> Apply dynamic programming...

1. Can find opt value for [1, ..., n] using only [1, .., j] with j < n.
2. Need solution to [1, ..., j] for each j = 1 ,..., n.
3. Solve each of those starting with j = 1.
   > opt value for [1] = w_1
   > opt value for [1, ..., j] = max(
   opt value for [1, ..., j-1],
   w_j + opt value for [1, ..., i] where $e_i \leq s_j$)

> Only n sub-problems

> Can solve all in O(n log n) time

Weighted Interval Scheduling

> Sort intervals by e_i in O(n log n) time

> Apply dynamic programming in O(n log n) time
  - only n sub-problems
  - can solve all in O(n log n) time
    > can actually solve all in O(n) time
    > binary searches are doing too much work (as in previous examples)
    > can optimize to O(n), but that doesn't improve overall run time

> As before, can get actual solution from the table

Outline for Today

> Weighted Interval Scheduling
> Max Sub-array Sum
> Document Layout in TeX
> Optimal Breakout Trades

Max Sub-array Sum

- **Problem**: Given an array A of integers,
  find max of A[i] + … + A[j-1] over all $0 \le i \le j \le n$
  - we allow i = j so that the sub-array A[i .. j-1] can be empty
  - note that A[i]'s can be **negative**
- Back to my favorite interview question…
  - brute force in $O(n^3)$ or $O(n^2)$
  - divide & conquer in $O(n \log n)$

Max Sub-array Sum

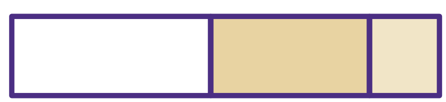
- Apply dynamic programming…
  - need to find a way to write the solution in terms of sub-problems
  - try the same approach as before…
- **Q**: does the opt solution include the last element?
  - If not, the answer is the optimal solution on A[0 .. n-2]
  - If yes, the answer is what??
    - A[n-1] + optimal solution on A[0 .. n-2] need not be a sub-array…

n-1

Max Sub-array Sum

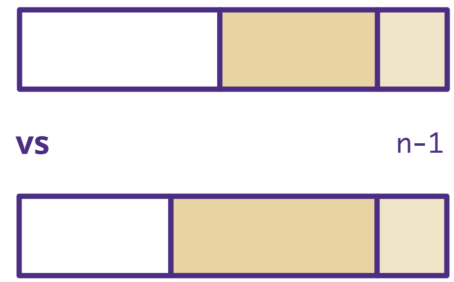
- Apply dynamic programming…
  - need to find a way to write the solution in terms of sub-problems
  - try the same approach as before…

- **Q**: does the opt solution include the last element?
  - If not, the answer is the optimal solution on A[0 .. n-2]
  - If yes, the answer is what??
    - A[n-1] + optimal solution on A[0 .. n-2] *ending at* n-2



n-1

Max Sub-array Sum



> Apply dynamic programming...
  – need to find a way to write the solution in terms of sub-problems
  – try the same approach as before...

> **Q**: does the opt solution include the last element?
  – If not, the answer is the optimal solution on A[0 .. n-2]
  – If yes, the answer is what??
    > A[n-1] + optimal solution ending at n-2
  – looks like we need two types of sub-problems:
    1. optimal solution over A[0 .. j-1]
    2. optimal solution over A[0 .. j-1] that end at A[j-1]

Max Sub-array Sum

> Looks like we need two types of sub-problems:
  1. optimal solution over A[0 .. j-1]
  2. optimal solution over A[0 .. j-1] that end at A[j-1]

> Sufficient to just solve sub-problems of type 2
  – every solution has to end *somewhere*
  – optimal value = max(opt value over A[0 .. j-1]) for j = 1 .. n

> Focus on just solving problems of type 2...

Max Sub-array Sum

only sub-arrays ending at n-1

- **Problem 2**: Given an array A of integers,
  find max of A[i] + ... + A[n-1] over all $0 \le i \le n$

- Apply dynamic programming...
- Find a way to write the solution in terms of sub-problems...

optimal substructure

- **Q**: does the opt solution include the last element?
  - if no, then opt value = 0
    - the only interval not including A[n-1] is the empty interval
  - if yes, then opt value = A[n-1] + opt value ending at n-2
    - every sub-array ending at n-1 is a subarray ending at n-2 + A[n-1]

Max Sub-array Sum

- **Q**: does the opt solution include the last element?
  - if yes, then opt value = A[n-1] + opt value ending at n-2
    - every sub-array ending at n-1 is a subarray ending at n-2 + A[n-1]
  - if no, then opt value = 0
    - the only interval not including A[n-1] is the empty interval

```
A = [31, -41, 59, 26, -53, 58, 97]
```

max (A[i] + … + A[n-1]) **with** i ≤ n-2
= max (A[i] + … + A[n-2] + A[n-1]) with i ≤ n-2
= max (A[i] + … + A[n-2]) with i ≤ n-2 + A[n-1]

Max Sub-array Sum

> Apply dynamic programming for opt sub-array ending at n-1...
   1. Can find opt value for A[0, ..., n-1] using only A[0, .., j-1] with j < n.
   2. Need opt value for A[0, ..., j-1] for each j = 1 ,..., n.
   3. Solve each of those starting with j = 1.
      > opt value for A[0 .. 0] = max(0, A[0])
      > opt value for A[0 .. j-1] = max(0, opt value for A[0 .. j-2] + A[j-1])

> Only n sub-problems
> Can solve all in O(n) time

Max Sub-array Sum

- Solve all sub-problems of type 2 in O(n) time
- Take maximum of these to solve original problem

- Better yet:
  - keep track of maximum as you go
  - no longer need to store entire array: just previous element and max so far

- Erasing your tracks will make you look smarter
  - solutions on web *do not* mention dynamic programming

Outline for Today

> Weighted Interval Scheduling
> Max Sub-array Sum
> Document Layout in TeX
> Optimal Breakout Trades

Paragraph Layout in TeX

> TeX is a document typesetting program
  - non-WYSIWYG
  - takes as input a description of the document
  - outputs a PDF (or similar) with the actual document

> Still widely used in mathematics and theoretical CS
  - mainly due to how well it formats equations
  - (partly just inertia)
  - generally considered to produce **beautiful** documents

Paragraph Layout in TeX

- TeX is a document typesetting program
  - non-WYSIWYG
  - takes as input a description of the document
  - outputs a PDF (or similar) with the actual document

- TeX program is one of the largest bug-free programs ever written
  - author, Don Knuth, is undoubtedly one of the best programmers in history
  - what counts as “bug-free”, however, is a matter of debate…

Paragraph Layout in TeX

- We will discuss paragraph layout
  - i.e., splitting words into lines
  - can choose to *stretch* or *shrink* space between words on a line
  - can break words using "-"

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus
elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dic-
tum gravida mauris. Nam arcu libero, nonummy eget, consectetuer
id, vulputate a, magna. Donec vehicula augue eu neque. Pellen-
tesque habitant morbi tristique senectus et netus et malesuada fames
ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem.
Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tel-
lus sit amet tortor gravida placerat. Integer sapien est, iaculis in,
pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices
bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulv-
inar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius
orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis
quis, diam. Duis eget orci sit amet orci dignissim rutrum.

- TeX uses a similar approach to break blocks into **pages**

Paragraph Layout in TeX

- Choose where splits should go between words to create lines
  - exponentially many options: 2^{n-1}, on a paragraph with n words
    - paragraphs with, e.g., 100 words would already be problematic
- Do so in order to minimize “badness” of the paragraph
  - overfull / underfull lines are infinitely bad
  - otherwise, badness = 100 (required stretch / shrink)^3
  - badness of paragraph is (essentially) **sum of line badness**
  - breaking between words with “-” has an extra penalty
  - other special cases… (e.g., last line is not stretched)
    - again, dynamic programming accommodates them without difficulty

Paragraph Layout in TeX

- Apply dynamic programming...
  - need to find a way to write the solution in terms of sub-problems
  - try the same approach as before...
- **Q**: does the opt solution include the last word?
  - obviously it does
  - the last word is always on the last time
  - need a better question...
- **Q**: What does the last line look like in opt solution?

Paragraph Layout in TeX

> Apply dynamic programming...
  - need to find a way to write the solution in terms of sub-problems
  - try thinking about the last line in the solution...

> **Q**: What does the last line look like in opt solution?
  - we know where it ends (at the last word)
  - only interesting question is where it *starts*
  - if it starts at word j, then cost is
    (opt value for words 1, ..., j-1) +
    badness of line with words j, ..., n
  - (actually only need to consider j up until last line becomes overfull)

Paragraph Layout in TeX

- Apply dynamic programming...
  - need to find a way to write the solution in terms of sub-problems
  - try thinking about the last line in the solution...

- **Q**: where does the last line start?
  - opt value for words 1, ..., n =
    max over j ≤ n of
    (opt value for words 1, ..., j-1) +
    badness of line with words j, ..., n
  - sub-problems again correspond to prefixes 1, ..., j-1
    - only n of them
  - BUT we need more than O(1) time to compute the formula

same <u>optimal substructure</u> as previous...
badness of line with j, ..., n
is common to all that split here
so opt must be opt of those too

Paragraph Layout in TeX

- Apply dynamic programming…
  1. Can find opt value for 1, .., n using only prefixes 1, …, j-1 with j ≤ n.
  2. Need opt value for 1, …, k for each k = 1, …, n.
  3. Solve each of those starting with k = 1.
     - opt value for 1 = badness of line [word 1]
     - opt value for 1, .., k = max over j ≤ k
       (opt value for 1, …, j-1) + (badness of line [word j, …, word k])

- Potentially O(n) per sub-problem, so O(n^2) time
  - in reality, there is a bound of, say, 40 words on a line
  - practical performance is O(n) [could be WYSIWYG today]

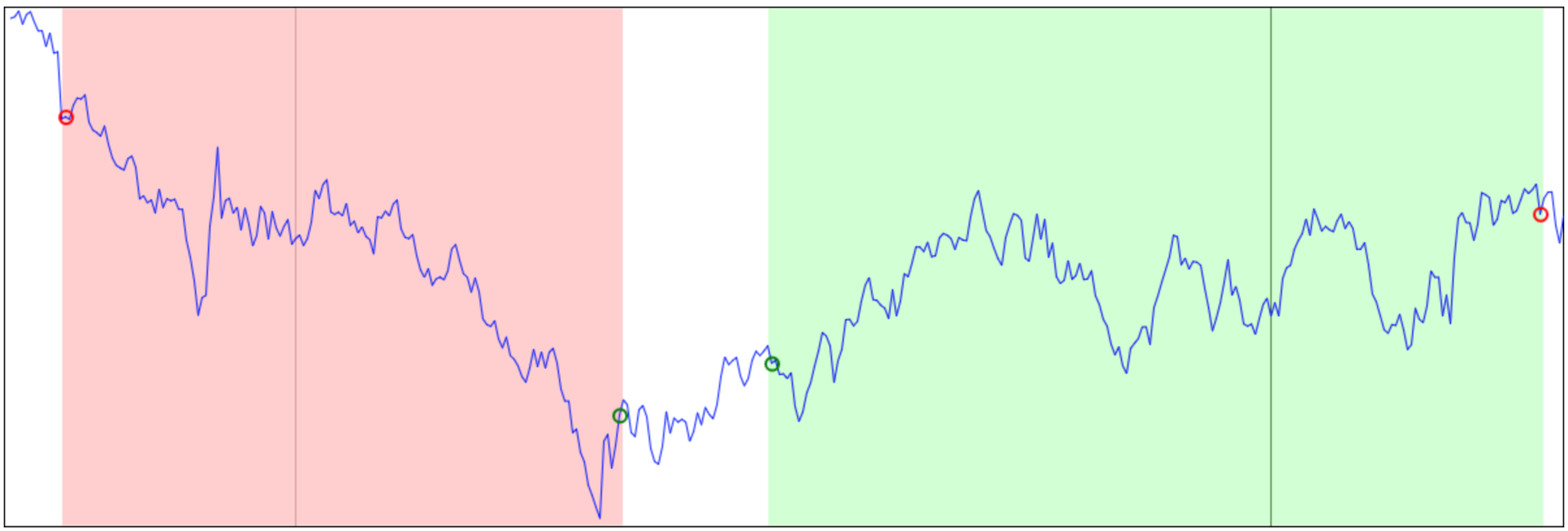
Outline for Today

> Weighted Interval Scheduling
> Max Sub-array Sum
> Document Layout in TeX
> Optimal Breakout Trades

Optimal Breakout Trades

- The usual advice is to buy low and sell high, but some trading strategies actually do the opposite!
- Goal: Figure out if that has any chance of being profitable.
- **Problem**: Given *future* prices, find the maximum profit that can be achieved from trades that only *buy on highs and sell on lows*.
  - can only buy when price is highest in 11 weeks
  - can only sell when price is lowest in 2 weeks
  - short selling allowed with reverse limits

Optimal Breakout Trades



Crude oil futures prices, 2015–16

Optimal Breakout Trades

- Exponentially many possible sequences of trades
  - brute force would not be feasible
- Apply dynamic programming...
  - to find optimal sub-structure, consider the last trade

Optimal Breakout Trades

- Apply dynamic programming...
  - to find optimal sub-structure, consider the last trade
- **Q**: What does the last trade look like in opt solution?
  - if it does not end by selling on the last day,
    then opt solution is the same as on prices 1 .. n-1
  - if it does end by selling on the last day,
    then opt value depends on where it buys...

Optimal Breakout Trades

- Apply dynamic programming...
  - to find optimal sub-structure, consider the last trade

- **Q**: What does the last trade look like in opt solution?
  - if it does not end by selling on the last day,
    then opt solution is the same as on prices 1 .. n-1
  - if it buys at time j and sells on the last day, then opt value =
    (opt value on 1 ... j-1) x (1 + percent change from j to n)
  - if it sells on the last day, then opt value = max over j < n of
    (opt value on 1 ... j-1) x (1 + percent change from j to n)

optimal substructure
(all are
multiplied by the
same number)

Optimal Breakout Trades

- Apply dynamic programming…
  1. Can find opt value for 1, .., n using only prefixes 1, …, j with j < n.
  2. Need opt value for 1, …, k for each k = 1, …, n.
  3. Solve each of those starting with k = 1.
     - opt value for 1 = 1 (can't sell until we buy)
     - opt value for 1, .., k = max(
       opt value for 1, …, k-1,
       max over j < k **of** (opt value for 1, …, j-1) x (1 + percent change from j to k)

- Potentially O(n) per sub-problem, so $O(n^2)$ time
  - can optimize further, but still $O(n^2)$ in worst case