1 Text Justification

Recall from lecture: split text into "good" lines

- obvious (MS Word/Open Office) algorithm: put as many words that fit on first line, repeat
- but this can make very bad lines

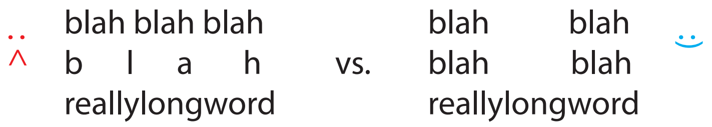
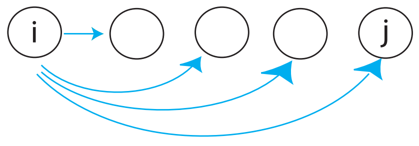


Figure 1: Good vs. Bad Text Justification.

- Define badness(i, j) for line of words$[i : j]$.
  For example, ∞ if total length $>$ page width, else (page width $-$ total length)^3.
- goal: split words into lines to $\min \sum$ badness

1. subproblem = min. badness for suffix words$[i :]$
   $\implies$ # subproblems $= \Theta(n)$ where $n =$ # words
2. guessing = where to end first line, say $i : j$
   $\implies$ # choices $= n - i = O(n)$
3. recurrence:
   - DP[i] = min(badness $(i, j) + DP[j]$ for j in range $(i + 1, n + 1)$)
   - $DP[n] = 0$
     $\implies$ time per subproblem $= \Theta(n)$
4. order: for $i = n, n - 1, \ldots, 1, 0$
   total time $= \Theta(n^2)$
5. solution $= DP[0]$



badness(i,j)

Figure 2: DAG.

2 5 Easy Steps to Dynamic Programming

1. define subproblems — count # subproblems
2. guess (part of solution) — count # choices
3. relate subproblem solutions — compute time/subproblem
4. recurse + memoize — time = time/subproblem · # subproblems
   OR build DP table bottom-up
   check subproblems acyclic/topological order
5. solve original problem: = a subproblem
   OR by combining subproblem solutions — $\Longrightarrow$ extra time

Examples:	Fibonacci	Shortest Paths
subprobs:	F_k for $1 \leq k \leq n$	$\delta_k(s, v)$ for $v \in V$, $0 \leq k < \lvert V \rvert$ = min $s \to v$ path using $\leq k$ edges
# subprobs:	n	V^2
guess:	nothing	edge into v (if any)
# choices:	1	$\text{indegree}(v) + 1$
recurrence:	$F_k = F_{k-1} + F_{k-2}$	$\delta_k(s,v) = \min\{\delta_{k-1}(s,u) + w(u,v) \mid (u,v) \in E\}$
time/subpr:	$\Theta(1)$	$\Theta(1 + \text{indegree}(v))$
topo. order:	for $k = 1, \ldots, n$	for $k = 0, 1, \ldots \lvert V \rvert - 1$ for $v \in V$
total time:	$\Theta(n)$	$\Theta(VE)$ + $\Theta(V^2)$ unless efficient about indeg. 0
orig. prob.:	F_n	$\delta_{\lvert V \rvert - 1}(s, v)$ for $v \in V$
extra time:	$\Theta(1)$	$\Theta(V)$

3 Perfect-Information Blackjack

Suppose that we want to play Blackjack against a dealer (with no other players). Suppose, in addition, that we have x-ray vision that allows us to see the entire deck $(c_0, c_1, ..., c_{n-1})$. As in a casino, the dealer will use a fixed strategy that we know ahead of time (stand-on-17), and that we area allowed to make \$1 bets (so with each round, we can either win \$1, lose \$1, or tie and make no profits and no losses). How do we maximize our winnings in this game? When should we hit or stand?

Let's use dynamic programming to try to come up with the best sequence of moves for us to make so that we will be able to earn as much as we can. Our approach will be as follows. We want to guess when to hit and when to stand for some suffix of cards starting at index i.

Let i be the number of cards we've already played, and BJ(i) the best play using the remaining cards $(c_i, c_{i+1}, ..., c_{n-1})$. These BJ$(i)$s will be our subproblems. How many of them are there? Since i spans from 0 to n, we have $O(n)$ subproblems.

What are our guesses? We have to guess how many times the player hits (requests another card). There are $O(n)$ choices for this guess. Here's a detailed recurrence:

```
BJ(i):
        if n-1 < 4: return 0, since there are not enough cards
        for  p in range(2, n-i-2): # number of cards taken
                # player's cards by deal order (player, then dealer, then player)
                player = sum(c_i, c_{i+2}, c_{i+4:i+p+2})
                if player > 21: # bust
                        options.append(-1 + BJ(i+p+2))
                        break
                for d in range(2, n-i-p):
                        dealer = sum(c_{i+1}, c_{i+3}, c_{i+p+2:i+p+d})
                        if dealer >= 17: break
                if dealer > 21: dealer = 0 # bust
                options.append(cmp(player, dealer) + BJ(i+p+d))
        return max(options)
```

Finding the player and dealer sums each takes $O(n)$. Looping over p takes $O(n)$. Therefore, the whole thing takes $O(n^2)$ time.

We can also construct a DAG for this problem as we have in the past for other problems. The graph can be defined as follows: costruct a vertex for each sub-problem, construct edges appropriately connecting the sub-problems (enumerate dependencies with edges). In this case, an edge is created for each subproblem with a greater i value (as seen in the recurrence where we append to options). The edge weights denote the outcome (-1, 0, 1). We will then try to find the longest path, maximizing our winnings (recall that the problem of finding the longest path on a DAG is equivalent to the shortest path problem on a DAG with negated edge weights, which we can solve in linear time).

4 Parent Pointers

To reconstruct the answer, we need to maintain information in addition to the value we are maximizing. There are many ways to do this, but here we discuss the parent pointers approach (which you may recall from numerous shortest path algorithms earlier this year). Instead of memoizing the result **value** in the table, we memoize a pair `(value, parent)`, where `parent` is the parent pointer to the chosen prior decision in the game tree. This means that in the table, the answer to the entire problem will be of the form `(result, parent_pointer)`, allowing us to follow the parent pointer to reconstruct the solution sequence (the last element of the sequence will have no parent).

5 Longest Increasing Subsequence

The longest increasing subsequence problem is a problem where we want to find a longest subsequence of a given sequence in which the subsequence elements are in sorted order. Note that a subsequence can skip elements in the original sequence. We can solve this using dynamic programming as well.

Wikipedia talks about the Van der Corput sequence. This is a sequence that, for any real number in $[0, 1]$, contains a subsequence that converges to that number. It's constructed by reversing the base n representation of the sequence of natural numbers (1, 2, 3, ...). For example, in base 10, the sequence begins with 0.1, 0.2, 0.3, ..., 0.9, 0.01, 0.11, 0.21, ...

The binary Van der Corput sequence begins with
0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15, ...

Two longest-increasing subsequences in this sequence are
0, 2, 6, 9, 13, 15.

0, 4, 6, 9, 11, 15.

To solve this using dynamic programming, we want to go through the sequence in order, keeping track of the longest increasing subsequence found so far. Let $a_1, a_2, ..., a_i$ denote the sequence, and define b_i to be the length of the longest subsequence ending with a_i. Now we can build each subproblem b_i as follows: find all j such that $j < i$, and $a_j <= a_i$, and collect them in a set S. Then b_i is simply $max b_j | j \in S + 1$ if $S \neq \emptyset$, and 1 if S is empty. After finding all n values for b, we are able to traverse the entire array and choose the largest. We use parent pointers to reconstruct the actual sequence fo the elments of the longest increasing subsequence in a

More detail here:
`http://www.algorithmist.com/index.php/Longest_Increasing_Subsequence`