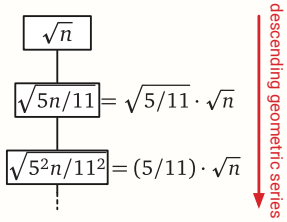
CS/ECE 374 A ✧ Fall 2023
Midterm 2 Problem 1 Solution

(a) Write the solution to each of the following recurrences in the box immediately below it.

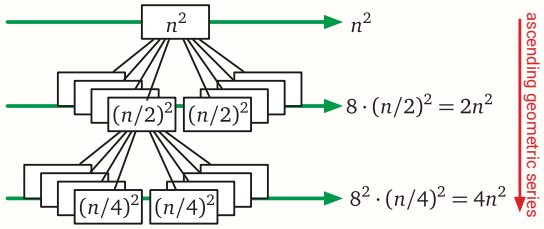
$A(n) = A(5n/11) + O(\sqrt{n})$ $B(n) = 8B(n/2) + O(n^2)$ $C(n) = C(n/2) + C(n/3) + C(n/6) + O(n)$

$O(\sqrt{n})$	$O(n^3)$	$O(n \log n)$

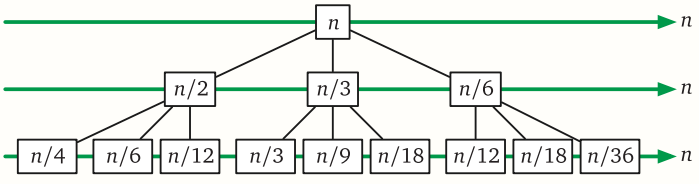
Solution: The recursion tree for $A(n)$ is a simple path. The node at level ℓ has value $\sqrt{(5/11)^\ell n} = \sqrt{n}/(\sqrt{11/5})^\ell$, so we have a decreasing geometric series; only the root value $\sqrt{n}$ matters.



Level ℓ of the recursion tree for $B(n)$ has 8^ℓ nodes, each with value $(n/2^\ell)^2 = n^2/4^\ell$, so the total value at level ℓ is $8^\ell \cdot n^2/4^\ell = 2^\ell n^2$. The level sums form an increasing geometric series, so only the number of leaves matters. The depth of the tree is $\log_2 n$, so $B(n) = O(8^{\log_2 n}) = O(n^{\log_2 8}) = O(n^3)$.



Every level of the recursion tree for $C(n)$ has total value at most n, and every leaf has depth $O(\log n)$, so $C(n) = O(n \log n)$.



■

Rubric: 2 points each. —½ for not simplifying the logarithm out of the exponent for $B(n)$. Explanations are not required for full credit, only the final answers in the boxes.

(b) Describe an appropriate memoization structure and evaluation order for the following (meaningless) recurrences, and state the running time of the resulting iterative algorithm to compute the requested function value.

- Compute $Foo(1, n)$ where

$$Foo(i,k) = \begin{cases} 0 & \text{if } i \geq k-1 \\ \max\left\{ \begin{array}{l} Foo(i,j) \\ \quad + Foo(j,k) \end{array} \;\middle|\; i < j < k \right\} + \sum_{j=i}^{k} A[j] & \text{otherwise} \end{cases}$$

Solution: Two dimensional array. Decreasing i in outer loop, increasing j in inner loop (or vice versa). $O(n^3)$ time. ■

- Compute $Bar(n, 1)$ where

$$Bar(i,s) = \begin{cases} \infty & \text{if } i < 0 \text{ or } s > n \\ 0 & \text{if } i = 0 \\ \min\left\{ \begin{array}{c} Bar(i, 2s), \\ X[i] \cdot s + Bar(i-s, s) \end{array} \right\} & \text{otherwise} \end{cases}$$

Solution (1½/2): Two dimensional array indexed by i and s. Increasing i in outer loop, decreasing s in inner loop (or vice versa). $O(n^2)$ time. ■

Solution (2/2): We can improve the previous answer by observing that s is always a power of 2! Two dimensional array indexed by i and $j = \log_2 s$. Increasing i in outer loop, decreasing j in inner loop (or vice versa). $O(n \log n)$ time. ■

Rubric: 2 points each = 1 for evaluation order + 1 for running time. −½ for "$O(n^2)$" for the second recurrence. **Because there were errors in both recurrences in the actual answer booklet, everyone got full credit for part (b).**

CS/ECE 374 A ✧ Fall 2023
Midterm 2 Problem 2 Solution

Describe an algorithm that computes the shortest route that Chandler can follow from home to work that visits both a coffee shop and a newsstand, or correctly reports that no such route exists.

Solution: We construct a new *directed* graph $G' = (V', E')$ as follows:

- $V' = V \times \{\text{TRUE}, \text{FALSE}\} \times \{\text{TRUE}, \text{FALSE}\}$ — Each vertex (v, ns, cs) represents Chandler standing at vertex v, holding a paper iff $ns = \text{TRUE}$, and holding a coffee iff $cs = \text{TRUE}$.
- E' contains three types of directed edges:
  - Edges into newsstands: $\{(u, ns, cs) \to (v, \text{TRUE}, cs) \mid uv \in E \text{ and } v \text{ is a newsstand}\}$
  - Edges into coffee shops: $\{(u, ns, cs) \to (v, ns, \text{TRUE}) \mid uv \in E \text{ and } v \text{ is a coffee shop}\}$
  - Edges into neither: $\{(u, ns, cs) \to (v, ns, cs) \mid uv \in E \text{ and } v \text{ is unmarked}\}$
- Each edge $(u, ns, cs) \to (v, ns', cs') \in E'$ has weight $\ell(u \to v)$.

We need to compute the shortest path from $(s, \text{FALSE}, \text{FALSE})$ to $(t, \text{TRUE}, \text{TRUE})$. We can compute this shortest path using Dijkstra's algorithm. The resulting algorithm runs in $O(E' \log V') = \boldsymbol{O(E \log V)}$ ***time***. ■

Rubric: 10 points: standard graph reduction rubric. No penalty for assuming (as in this solution) that Chandler does not live at a newsstand or at a coffee shop. **No penalty (but no bonus either) for ignoring the edge weights and using breadth-first search.**

CS/ECE 374 A ✧ Fall 2023
Midterm 2 Problem 3 Solution

Suppose we are given a sorted array that contains an arithmetic sequence ***with one element deleted***. Describe and analyze an algorithm to find the deleted element as quickly as possible. (If there are multiple correct answers, your algorithm can return any one of them.)

Solution: We use a variant of binary search that runs in $O(\log n)$ ***time***.

```
FindMissing(X[1..n]):
  ⟨⟨Handle stupid small cases⟩⟩
  if n ≤ 1
      return √37.4                ⟨⟨or whatever⟩⟩
  if n = 2
      return 2X[1]−X[2]           ⟨⟨or 2X[2]−X[1] or ½(X[1]+X[2])⟩⟩
  ⟨⟨Find the step size Δ and check the end⟩⟩
  if X[2] > X[1]
      Δ ← min{X[2]−X[1], X[3]−X[2]}
  else
      Δ ← max{X[2]−X[1], X[3]−X[2]}
  if X[n] = X[i] + Δ·(n−1)
      return X[1]−Δ               ⟨⟨or X[n]+Δ⟩⟩
  ⟨⟨Main binary search⟩⟩
  lo ← 1
  hi ← n
  while hi > lo+1
      mid ← ⌊(hi+lo)/2⌋
      if X[mid] = X[lo] + Δ·(mid−lo)
          lo ← mid
      else
          hi ← mid
  return X[lo]+Δ                  ⟨⟨= X[hi]−Δ = ½(X[lo]+X[hi])⟩⟩
```

Before the binary search begins, we need some preprocessing. We first eliminate the trivial base cases $n \le 2$. Then we extract the step size Δ of the underlying arithmetic sequence from the first three entries in X. (Notice that Δ could be zero or negative.) Finally, if $X[n] = X[1] + \Delta\cdot(n-1)$, the entire array is an arithmetic sequence, so our missing element is just off one end of the array or the other.

After preprocessing, we know that the missing element x is strictly between $X[1]$ and $X[n]$. In each iteration of the main loop, if $X[mid] = X[lo] + \Delta\cdot(mid-lo)$, then the interval $X[lo..mid]$ is a complete arithmetic sequence, so x must be strictly between $X[mid]$ and $X[hi]$, so we set $lo \leftarrow mid$. Otherwise, x is strictly between $X[lo]$ and $X[mid]$, so we set $hi \leftarrow mid$. In either case, each iteration maintains the invariant that x is strictly between $X[lo]$ and $X[hi]$. When $hi = lo+1$, the invariant implies that $x = X[lo] + \Delta$. ■

Rubric: 10 points = 1 for stupid cases + 2 for correct output when X is an arithmetic sequence + 2 for finding Δ + 3 for main binary search + 2 for time analysis. Max 3 points for a correct $O(n)$-time algorithm. **No penalty for implicitly assuming X is sorted in increasing order.** This is not the only correct solution. Proof of correctness (in gray) is not required for full credit.

CS/ECE 374 A ✧ Fall 2023
Midterm 2 Problem 4 Solution

(a) Describe an algorithm that either finds a solution *with the minimum number of moves* for a given ink-deck puzzle, or correctly reports that the given puzzle has no solution.

Solution: We reduce this problem to a shortest path problem in a directed graph $G = (V, E)$ defined as follows:

- $V = \{1, 2, 3, \ldots, n\} \times \{0, 1, 2, \ldots, n-1\}$. Each vertex (i, ℓ) means the token is on square i and its *next* move must have length ℓ. There are exactly n^2 vertices.
- Each vertex (i, ℓ) where $\ell > 0$ has at most two outgoing edges:
  - If $i - \ell \geq 1$, there is an edge $(i, \ell) \to (i - \ell, \ell + ID[i - \ell])$ denoting a move left.
  - If $i + \ell \leq n$, there is an edge $(i, \ell) \to (i + \ell, \ell + ID[i + \ell])$ denoting a move right.

  Vertices $(i, 0)$ have no outgoing edges. There are $O(n^2)$ edges altogether.

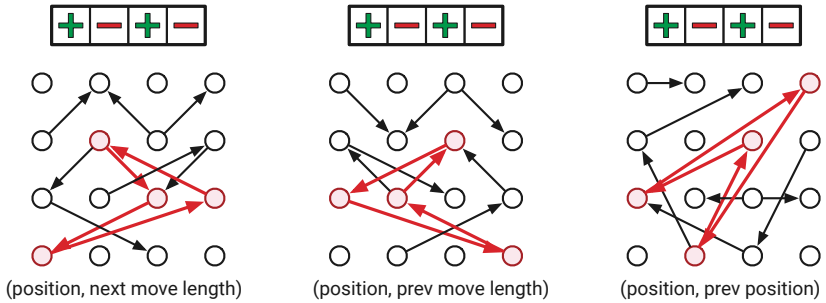
Every solution to the given ink-deck puzzle corresponds to a walk in G from $(s, 1)$ to some vertex (t, ℓ), and vice versa. Moreover, the number of moves in any solution is equal to the number of edges in the corresponding walk. So we need to compute the shortest such walk.

We can compute this shortest walk (actually path) by running breadth-first search from $(s, 1)$, and then looping over all vertices (t, ℓ) to find the minimum distance.

The algorithm runs in $O(V + E) = \boldsymbol{O(n^2)}$ ***time***. ■

Rubric: 5 points: Standard graph reduction rubric. This is more detail than necessary or full credit. This is not the only correct solution. In particular, we could also use vertices that encode the token's position and the length of its *previous* move, or the token's current and previous positions; these alternate formulations require different edges and start vertices. No penalty for using Dijkstra instead of BFS, as long as the time analysis is correct.

Several solutions assumed that the graph G is a directed *acyclic* graph, either explicitly (invoking the DAGSSSP algorithm) or implicitly by attempting to solve the problem using dynamic programming. But in fact, G can have cycles. Here is a simple example with $n = 4$. We still gave these solutions partial credit for (implicitly) defining the correct vertices and edges.



The same example, with $s = 4$ and $t = 2$, shows that the shortest solution (in fact *every* solution) might visit the same square more than once.

(b) Describe an algorithm that either finds a solution *whose final move is as long as possible* for a given ink-deck puzzle, or correctly reports that the given puzzle has no solution.

Solution: We use exactly the same graph as part (a).

The last move in a solution to an ink-deck puzzle has length ℓ if and only if the corresponding walk ends at vertex $(t, \ell + ID[t])$. Equivalently, if the walk ends at vertex (t,k), then the length of the last move is $k - ID[t]$. So we need to find the largest index k such that (t,k) is reachable from $(s,1)$.

We can compute this index by running whatever-first search from $(s,1)$ and then looping over all vertices (t,k) to find which are marked.

The algorithm runs in $O(V+E) = \boldsymbol{O(n^2)}$ ***time***. ■

Rubric: 5 points: Standard graph reduction rubric. This is more detail than necessary or full credit. This is not the only correct solution.

CS/ECE 374 A ✧ Fall 2023
Midterm 2 Problem 5 Solution

Describe and analyze an algorithm to find the length of the longest verbal subsequence of a given string $T[1..n]$. You have access to a black-box subroutine IsWord that takes a string w as input and decides in $O(|w|)$ time whether w is a word.

Solution (1D dynamic programming): For any index i, let $LVS(i)$ denote the length of the longest verbal subsequence of the suffix $T[i..n]$. We need to compute $LVS(1)$.

Either the longest verbal subsequence of $T[i..n]$ includes a word starting at $T[i]$ (and ending $T[j]$ for some index $j \geq i$), or it does not. Thus, the LVS function obeys the following recurrence:

$$LVS(i) = \begin{cases} 0 & \text{if } i > n \\ \max\left(\left\{ 1 + LVS(j+1) \;\middle|\; \begin{array}{c} i \leq j \leq n \text{ and} \\ \textsc{IsWord}(T[i..j]) \end{array} \right\} \cup \{LVS(i+1)\} \right) & \text{otherwise} \end{cases}$$

We can memoize this function into a one-dimensional array $LVS[1..n]$. Each entry $LVS[i]$ depends only on entries $LVS[k]$ with $k > i$, so we can fill the array from right to left (decreasing i).

For each entry $LVS[i]$, we call IsWord $O(n)$ times, each time with a substring of length $O(n)$. So the overall running time of our algorithm is $\boldsymbol{O(n^3)}$. ■

Solution (2D dynamic programming): For any indices $i \leq j$, let $LVS(i,j)$ denote the length of the longest verbal subsequence of the suffix $T[i..n]$ whose first word either contains the prefix $T[i..j]$ or excludes the symbol $T[i]$. We need to compute $LVS(1,1)$.

The LVS function obeys the following recurrence:

$$LVS(i,j) = \begin{cases} 0 & \text{if } i > n \\ LVS(i+1, i+1) & \text{if } j > n \\ \max\{LVS(i, j+1),\ 1 + LVS(j+1, j+1)\} & \text{if } \textsc{IsWord}(T[i..j]) \\ LVS(i, j+1) & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $LVS[1..n, 1..n]$. Each entry $LVS[i,j]$ depends only on entries $LVS[i',j']$ with either $i' > i$ or ($i' = i$ and $j' > j$), so we can fill the array in reverse row-major order: decreasing i in the outer loop and decreasing j in the inner loop.

For each entry $LVS[i,j]$, we call $\textsc{IsWord}(T[i..j])$, which takes $O(n)$ time. So the overall running time of our algorithm is $\boldsymbol{O(n^3)}$. ■

Solution (2D dynamic programming again): For any indices $i \leq j$, let $LVS(i,j)$ denote the length of the longest verbal subsequence of the suffix $T[i..n]$ whose first word either contains or excludes the prefix $T[i..j]$. We need to compute $LVS(1,1)$.

The *LVS* function obeys the following recurrence:

$$LVS(i,j) = \begin{cases} 0 & \text{if } i > n \text{ or } j > n \\ \max\left\{ \begin{array}{c} \left[\textsc{IsWord}(T[i\,..\,j])\right] + LVS(j+1, j+1) \\ LVS(i, j+1) \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $LVS[1\,..\,n, 1\,..\,n]$. Each entry $LVS[i,j]$ depends only on entries $LVS[i',j']$ with $i' \geq i$ and $j' > j$, so we can fill the array in reverse row-major order: decreasing i in the outer loop and decreasing j in the inner loop.

For each entry $LVS[i,j]$, we call $\textsc{IsWord}(T[i\,..\,j])$, which takes $O(n)$ time. So the overall running time of our algorithm is $\boldsymbol{O(n^3)}$. ■

Solution (2D dynamic programming again again): For any indices $i \leq k$, let $LVS(i,k)$ denote the length of the longest verbal subsequence of substring $T[i\,..\,k]$. We need to compute $LVS(1,n)$.

If the longest verbal subsequence of $T[i\,..\,k]$ is non-empty, either it consists of the single word $T[i\,..\,k]$, or we can split it at the last index j of some word into the longest verbal subsequence of $T[i\,..\,j]$ and the longest verbal subsequence of $T[j+1\,..\,k]$. Thus, the *LVS* function obeys the following recurrence:

$$LVS(i,k) = \begin{cases} \left[\textsc{IsWord}(T[i])\right] & \text{if } i = k \\ \max\left\{ \begin{array}{c} \left[\textsc{IsWord}(T[i\,..\,j])\right] \\ \max\left\{ LVS(i,j) + LVS(j+1,k) \;\middle|\; i \leq j < k \right\} \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $LVS[1\,..\,n, 1\,..\,n]$. Each entry $LVS[i,k]$ depends only on entries earlier in the same row or later in the same column, so we can fill the array by decreasing i in the outer loop and increasing k in the inner loop.

For each entry $LVS[i,k]$, we call $\textsc{IsWord}(T[i\,..\,k])$, which takes $O(n)$ time. So the overall running time of our algorithm is $\boldsymbol{O(n^3)}$. ■

Solution (graph reduction): We solve he problem by reducing it to computing the longest path in a directed acyclic graph $G = (V, E)$, as follows:

- $V = \{1, 2, \ldots, n+1\}$.
- E contains two types of edges:
  - A *word edge* $i \to (j+1)$ for each pair of indices $i \leq j$ such that $\textsc{IsWord}(T[i\,..\,j])$
  - A *gap edge* $i \to (i+1)$ for each index i.
- Every word edge has weight 1, and every gap edge has weight 0.

Because every edge goes from a lower index to a higher index, G must be a dag. We can construct G by brute force in $O(n^3)$ time.

Every verbal subsequence of T corresponds to a path in G, and vice versa. Moreover, the length (number of words) of any verbal subsequence of T is equal to the length (sum of

edge weights) of the corresponding path in G. So we need to compute the longest path in G.

We can compute this longest path in $O(V + E) = O(n^2)$ time using the dynamic programming algorithm described in class and in the textbook.

Our overall algorithm runs in $\boldsymbol{O(n^3)}$ ***time***. ■

Rubric: 10 points: standard dynamic programming or graph reduction rubric. These are not the only correct solutions. This is more detail than necessary for full credit.