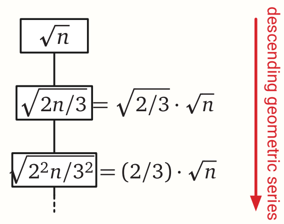
CS/ECE 374 A ✧ Fall 2023
Conflict Midterm 2 Problem 1 Solution

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

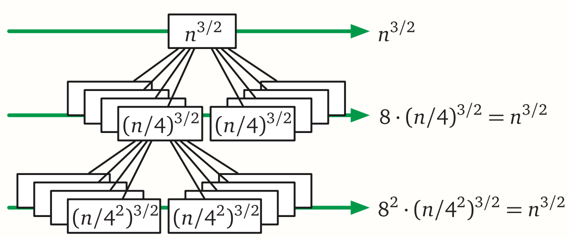
$A(n) = A(2n/3) + O(\sqrt{n})$ $\quad$ $B(n) = 8B(n/4) + O(n^{3/2})$ $\quad$ $C(n) = C(n/2) + C(n/3) + O(n)$

$\boxed{O(\sqrt{n})}$ $\quad$ $\boxed{O(n^{3/2}\log n)}$ $\quad$ $\boxed{O(n)}$

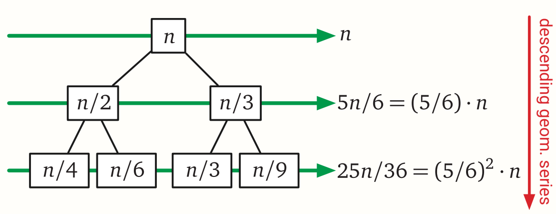
Solution: The recursion tree for $A(n)$ is a simple path. The node at level ℓ has value $\sqrt{(2/3)^\ell n} = \sqrt{n}/(\sqrt{3/2})^\ell$, so we have a decreasing geometric series; only the root value $\sqrt{n}$ matters.



Level ℓ of the recursion tree for $B(n)$ has 8^ℓ nodes, each with value $(n/4^\ell)^{3/2} = n^{3/2}/8^\ell$, so the total value at every level is $n^{3/2}$. The depth of the tree is $O(\log_2 n)$.



The root of the recursion tree for $C(n)$ has value n; the two children of the root sum to $n/2+n/3 = 5n/6$; the four grandchildren of the root sum to $n/4+n/6+n/6+n/9 = 9n/36 + 6n/36 + 6n/36 + 4n/36 = 25n/36$. More generally, the sum of the nodes at level ℓ is *at most* $(5/6)^\ell n$. We have a decreasing geometric series; only the root value n matters.



■

Rubric: 2 points each. 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 $Pleb(n, 1)$ where

$$Pleb(i,k) = \begin{cases} i & \text{if } k \leq 0 \\ k & \text{if } i > n \\ \max \begin{Bmatrix} i + k + Pleb(i-1, k+1) \\ i + Pleb(i-1, k) \\ k + Pleb(i, k+1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

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

- Compute $Nom(1, n)$ where

$$Nom(i,k) = \begin{cases} 0 & \text{if } k = i \\ (X[k] - X[i]) + \min \left\{ \begin{matrix} Nom(i,j) \\ +Nom(j,k) \end{matrix} \;\middle|\; i < j < k \right\} & \text{otherwise} \end{cases}$$

> **Solution:** Two dimensional array indexed by i and k. Decreasing i in the outer loop, increasing k in the inner loop (or vice versa). $O(n^3)$ time. ■

Rubric: 2 points each = 1 for evaluation order + 1 for running time. **Because there were errors in both recurrences in the answer booklet for the regular exam, everyone got full credit for part (b).**

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

Suppose you are given a directed graph G in which every edge is either red or blue, and a subset of the vertices are marked as *special*. A walk in G is *legal* if color changes happen only at special vertices.

Describe and analyze an algorithm that either returns the length of the shortest legal walk in G from vertex s to vertex t, or correctly reports that no such walk exists.

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

- $V' = V \times \{\text{red}, \text{blue}\}$. Each vertex (v, c) indicates that we are located at vertex v and the next edge we traverse must have color c. There are $2V$ vertices in V'.
- There are four types of edges in E':
  - Normal red edges: $\big\{(u, \text{red}) \to (v, \text{red}) \bigm| u \to v \in E \text{ is red}\big\}$
  - Normal blue edges: $\big\{(u, \text{blue}) \to (v, \text{blue}) \bigm| u \to v \in E \text{ is blue}\big\}$
  - Special red edges: $\big\{(u, \text{red}) \to (v, \text{blue}) \bigm| u \to v \in E \text{ is red and } v \in V \text{ is special}\big\}$
  - Special blue edges: $\big\{(u, \text{blue}) \to (v, \text{red}) \bigm| u \to v \in E \text{ is blue and } v \in V \text{ is special}\big\}$

  There are at most $2E$ edges in E'.
- Edges in E' do not have weights.
- Every legal walk in G corresponds to a walk in G' and vice versa. Moreover, the length (number of edges) in any walk in G is equal to length the corresponding walk in G'. Thus, we are looking for the shortest walk in G' from either (s, red) or (s, blue) to either (t, red) or (t, blue).
- We can find these four shortest paths using $O(1)$ calls to breadth-first search, in $O(V' + E') = \boldsymbol{O(V + E)}$ ***time***.

■

Rubric: 10 points: standard graph reduction rubric. **No penalty for using Dijkstra in $O(E \log V)$ time instead of breadth-first search.** This is not the only correct solution. In particular:

- We could define special edges $(u, \text{red}) \to (v, \text{blue})$ and $(u, \text{blue}) \to (v, \text{red})$ for each edge $u \to v$ *leaving* a special vertex u, instead of entering a special vertex v.
- We could define special edges $(v, \text{red}) \to (v, \text{blue})$ and $(v, \text{blue}) \to (v, \text{red})$ at each special vertex v. But to preserve length, we need to give these edges weight 0. Computing shortest paths in this graph in $O(V + E)$ time is still possible, but more complicated.

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

(a) Describe an algorithm that, given an ink-deck puzzle, either finds a solution whose *total* length is as *small* as possible, or correctly reports that there is 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 $(i, \ell) \rightarrow (i - \ell, \ell + ID[i - \ell])$ denoting a move left.
  - If $i + \ell \leq n$, there is an edge $(i, \ell) \rightarrow (i + \ell, \ell + ID[i + \ell])$ denoting a move right.

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

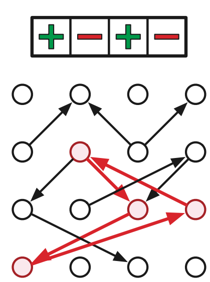
Every solution to the given ink-deck puzzle corresponds to a walk in G from $(s, 1)$ to some vertex (t, ℓ), and vice versa. Moreover, the total length of the moves in any solution is equal to the length (total weight) of the corresponding walk. So we need to compute the shortest such walk.

We can compute this shortest walk (actually path) by running Dijkstra's algorithm from $(s, 1)$, and then looping over all vertices (t, ℓ) to find the minimum distance.

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

Rubric: 5 points: Standard graph reduction rubric. This is more detail than necessary for full credit. This is not the only correct solution. In particular, we could interpret each vertex (i, ℓ) to indicate that the token is at square i and its *previous* move had length ℓ; this interpretation requires different edges, different edge weights, and a different start vertex. **No penalty for ignoring edge weights and using BFS instead of Dijkstra.** Max 4 points for a correct algorithm that runs in $O(n^3)$ time; max 3 points for an algorithm that runs in $O(n^4)$ time; scale partial credit.

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 $n = 4$:



The same example, with $s = 4$ and $t = 2$, shows that the shortest sequence of moves can revisit the same square more than once.

(b) Describe an algorithm that, given an ink-deck puzzle, either finds a solution whose *maximum* length is as *large* as possible, or correctly reports that there is no solution.

Solution (pure reachability): We use exactly the same graph G as part (a), but without the edge weights. To simplify the algorithm we add an artificial sink z with edges $(t,\ell)\to z$ for every ℓ.

A move from square i of length ℓ (that is, to either square $i+\ell$ or square $i-\ell$) appears in a solution to an ink-deck puzzle if and only if two reachability conditions are satisfied:

- Vertex (i,ℓ) is reachable from $(s,1)$.
- Vertex z is reachable from (i,ℓ).

Thus, we need to find the maximum index ℓ such that some vertex (i,ℓ) satisfies these two conditions.

We can find all vertices reachable from $(s,1)$ using a single whatever-first search in G. Similarly, we can find all vertices of G that can reach z using a single whatever-first search in the reversal of G. Finally, we can compute the maximum index ℓ by looping over all vertices by brute force, and considering only those that are marked by both searches.

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

Solution (via strong components): We start with the same graph G as part (a), but without the edge weights. To simplify the algorithm we add an artificial sink z with edges $(t,k)\to z$ for every k.

Let $[v]$ denote the strong component of vertex v in G. First we compute the meta-graph $[G] = ([V],[E])$ of G using either textbook algorithm in $O(V+E) = O(n^2)$ time. Each set $[v]$ is a vertex of the meta-graph $[G]$. For each meta-vertex $[v]$, we compute and store the maximum length $[v].max\ell = \max\{\ell \mid (i,\ell) \in [v] \text{ for some } i\}$. In particular, we define $[z].max\ell = 0$.

For any meta-vertex $[v]$, let $MaxMax\ell([v])$ denote the largest maximum-length of any meta-vertex on any path from $[v]$ to $[z]$. We need to compute $MaxMax\ell([(s,1)])$. This function obeys the following recurrence:

$$MaxMax\ell([v]) = \begin{cases} 0 & \text{if } [v] = [z] \\ \max\Big(\{[v].max\ell\} \cup \big\{MaxMax\ell([v]) \bigm| [v]\to[w]\big\}\Big) & \text{otherwise} \end{cases}$$

By memoizing into the meta-graph itself, we can compute $MaxMax\ell([v])$ for every meta-vertex $[v]$ in reverse topological order (that is, in postorder) in $O(V'+E') = O(V+E) = O(n^2)$ time.

The overall running time of the algorithm is $\boldsymbol{O(n^2)}$. ■

Rubric: 5 points: Standard graph reduction rubric (at least for the first solution). This is more detail than necessary or full credit. These are not the only correct solutions. No penalty for using BFS or DFS instead of WFS. Max 4 points for a correct algorithm that runs in $O(n^3)$ time (for example, running WFS in $rev(G)$ from each vertex (t, k)). Max 3 points for an algorithm that runs in $O(n^4)$ time (for example, running WFS from each vertex of G). Scale partial credit.

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

Suppose we are given a sorted array that contains an arithmetic sequence *with one element repeated once*. Describe and analyze an algorithm to find the deleted element as quickly as possible.

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

```
FindDupe(X[1..n]):
  if X[1] = X[2]
      return X[1]
  if X[n] = X[n−1]
      return X[n]
  Δ ← X[2] − X[1]
  ⟨⟨Main binary search⟩⟩
  lo ← 1
  hi ← n
  while hi > lo + 1
      mid ← ⌊(hi + lo)/2⌋
      if (X[mid] = X[mid + 1]) or (X[mid] = X[mid − 1])
          return X[mid]
      if X[mid] = X[lo] + Δ · (mid − lo)
          lo ← mid
      else
          hi ← mid
  destroy the universe
```

Before the binary search begins, we need a small amount of preprocessing. If the first two elements of X are equal, then either they are the only repeated element of a nontrivial arithmetic sequence, or every element of X has the same value. In either case, the repeated value is equal to $X[1]$. If $X[1]$ and $X[2]$ are not equal, the step size Δ of the underlying arithmetic sequence is their difference. (Notice that Δ could be negative.)

After preprocessing, we know that the repeated value x appears strictly between $X[1]$ and $X[n]$. In each iteration of the main loop, we first check if $X[mid]$ is the repeated value. If not, then if $X[mid] = X[lo] + \Delta \cdot (mid - lo)$, then the interval $X[lo \,..\, mid]$ is a complete arithmetic sequence, so x must appear 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 $X[hi]$. This invariant implies that the algorithm returns an answer before it destroys the universe; if $hi = lo + 1$, so the loop ends, the invariant implies that neither $X[lo]$ nor $X[hi]$ is the repeated value, so the repeated value must not exist! ■

Rubric: 10 points = 1 for stupid cases + 1 for correct output when X is constant + 2 for finding Δ + 4 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
Conflict Midterm 2 Problem 5 Solution

Describe and analyze an algorithm to find the length of the *longest substring* of T that can be split into words. 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: For any index i, let $LSP(i)$ denote the length of the Longest Splittable Prefix of the suffix $T[i\,..\,n]$. We need to compute $\max_i LSP(i)$.

Either the longest splittable prefix of $T[i\,..\,n]$ is empty, or it starts with a word $T[i\,..\,j]$ for some index $j \ge i$. Thus, the LSP function obeys the following recurrence (assuming $\max\varnothing = 0$):

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

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

For each entry $LSP[i]$, 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: For all indices $i < k$, let $Splittable(i,k) = \textsc{True}$ if the substring $T[i\,..\,k]$ can be partitioned into words, and $\textsc{False}$ otherwise. We need to compute the integer $\max\{k - i + 1 \mid Splittable(i,k)\}$.

$T[i\,..\,k]$ is splittable if and only if it is a single word, or it can be partitioned into two splittable substrings $T[i\,..\,j]$ and $T[j+1\,..\,k]$ at some index $i \le j < k$. Thus, the *Splittable* function obeys the following recurrence:

$$Splittable(i,k) = \begin{cases} \textsc{IsWord}(T[i\,..\,i]) & \text{if } i = k \\ \textsc{IsWord}(T[i\,..\,k]) \vee \displaystyle\bigvee_{j=i}^{k-1} \big(Splittable(i,j) \wedge Splittable(j+1,k)\big) & \text{otherwise} \end{cases}$$

(The second case is redundant if we define $\bigvee\varnothing = \textsc{False}$.)

We can memoize this function into a two-dimensional array $SplitTable[1\,..\,n, 1\,..\,n]$. Each entry $SplitTable[i,k]$ 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 (or vice versa). For each entry $SplitTable[i,k]$, we call IsWord once with a substring of length $O(n)$, so filling the table takes $O(n^3)$ time. Once the table is full, we compute $\max\{k - i + 1 \mid SplitTable[i,k]\}$ in $O(n^2)$ time by brute force. Altogether our algorithm runs in $\boldsymbol{O(n^3)}$ ***time***. ■

Solution: We reduce this problem to finding the longest weighted path in a dag $G = (V, E)$ as follows:

- $V = \{1, 2, 3 \ldots, n, n+1\}$
- $E = \{i \to j \mid 1 \le i < j \le n+1 \text{ and } \textsc{IsWord}(T[i\,..\,j-1])\}$.
- Each edge $i \to j$ has weight $j - i$.

G is acyclic because $i < j$ for every edge $i \to j$. In fact, the vertex numbering $1, 2, 3 \ldots, n, n+1$ is already a topological order for G.

Each directed edge in G represents a substring of T that is a single word, and each path in G represents a sequence of contiguous word substrings in T. More precisely, there is a path in G from vertex i to vertex k (which must have weighted length $k - i$) if and only if the substring $T[i\,..\,k-1]$ (which also has length $k - i$) is splittable into words. Thus, we need to compute the length of the longest path in G.

We can construct G by brute force in $O(n^3)$ time by calling $\textsc{IsWord}(T[i\,..\,j-1])$, for every pair of indices i and j such that $1 \le i < j \le n+1$, in arbitrary order. Then we can compute the length of the longest path in G by dynamic programming, as described in class and in the textbook, in $O(V + E) = O(n^2)$ time. The overall algorithm runs in ***$O(n^3)$ time***. ■

Non-solution: Several submissions described algorithms that solve a *different problem* than the one we asked. The most common different problems were:

- Find the length of the longest substring of T that is a ***single word***.
- Find the maximum total length of ***disjoint*** (but not necessarily contiguous) word substrings of T. For example, given the input string STURDYNAMICEXTRAPROGRAMBLE, this algorithm would return $6 + 4 + 4 + 6 = 20$ (for STURDY+MICE+TRAP+RAMBLE) instead of $7 + 5 + 7 = 19$ (for DYNAMIC+EXTRA+PROGRAM). This is the "longest verbal subsequence" problem from the regular midterm.
- Find the maximum total length of ***nested*** word substrings. For example, given the input string XMANSLAUGHTERCOOKIESX, this algorithm would return $5 + 8 + 12 = 25$ (for LAUGH+LAUGHTER+MANSLAUGHTER or AUGHT+LAUGHTER+MANSLAUGHTER) instead of $3 + 9 + 7 = 19$ (for MAN+SLAUGHTER+COOKIES).

We still gave partial credit for clear English descriptions and base cases that were consistent with correct algorithms. ♣

Rubric: 10 points: standard dynamic programming rubric. These solutions have more detail than necessary for full credit. These are not the only correct solutions. Max 8 points for an $O(n^4)$-time algorithm; max 6 points for an $O(n^5)$-time algorithm; scale partial credit.

No penalty for assuming all words must have length at least 4, even though that restriction only applied to the specific English examples.