CSE 101: Design and Analysis of Algorithms
Winter 2018

Homework 2 Solutions
Due: 02/03/2018, 11.59 PM

Exercise 1. DPV 2.4

Solution:

- Algorithm A: Algorithm A has running time:

$$T_A(n) = 5T_A(\frac{n}{2}) + \mathcal{O}(n)$$

  We solve this recurrence by applying the Master Theorem, with $T(n) = aT(\frac{n}{b}) + \mathcal{O}(n^d)$, we obtain $a = 5$, $b = 2$ and $d = 1$. We get $T_A(n) = O(n^{\log_2 5})$.

- Algorithm B: Algorithm B has running time:

$$T_B(n) = 2T_B(n-1) + \mathcal{O}(1)$$

  We solve this recurrence by expanding it and replacing O(1) by a constant c:

$$\begin{aligned} T_B(n) &= 2T_B(n-1) + c \\ &= 2(2\ T_B(n-2) + c) + c = 4T_B(n-2) + 3c \\ &= 4(2\ T_B(n-3) + c) + 3c = 8T_B(n-3) + 7c \end{aligned}$$

  The pattern is $T_B(n) = 2^k 2T_B(n-k) + c(2^k - 1)$. Setting $k = n-1$, we see that $T_B(n) = 2^n T(1) + c(2^{n-1} - 1) = O(2^n)$.

- Algorithm C: Algorithm C has running time:

$$T_C(n) = 9T_C(n/3) + O(n^2) \tag{1}$$

  We solve this recurrence by applying the Master Theorem, with $T(n) = aT(\frac{n}{b}) + \mathcal{O}(n^d)$, we obtain $a = 9$, $b = 3$ and $d = 2$. We get, $T_C(n) = \mathcal{O}(n^2 \log n)$.

- We choose Algorithm C as it has the smallest (asymptotic) running time.

Exercise 2. DPV 2.19

Solution:

(a) Time Complexity: The merge procedure takes time $\mathcal{O}(l_1 + l_2)$ to merge two arrays of size l_1 and l_2. In this strategy, we are performing the merge procedure $k-1$ times. In the j^{th} merge procedure, one of the arrays has size jn while the other has size n; therefore, the time taken by the j^{th} merge procedure is $\mathcal{O}((j+1)n)$. The total time taken is then $\mathcal{O}(2n + 3n + 4n + ... + kn) = \mathcal{O}(k^2 n)$.

(b) Let $multimerge(A_1, A_2, \cdots, A_k)$ denote the result of merging arrays $A_1, \cdots, A_k$, and let merge be the two-way merge operation discussed in the textbook. We can do a *multimerge* using divide and conquer:

$$\begin{aligned} multimerge(A_1, A_2, \cdots, A_k) = merge(&multimerge(A_1, A_2, \cdots, A_{k/2}) \\ &, multimerge(A_{\frac{k}{2}+1}, \cdots, A_k)) \end{aligned}$$

Let $T_n(k, n)$ denote the time taken to multimerge k arrays of length n. Then the recurrence for $T_n(k, n)$ is:

$$T_n(k, n) = 2T_n(\frac{k}{2}, n) + \mathcal{O}(kn)$$

This is because there are two recursive calls to *multimerge*, each taking time $T_n(\frac{k}{2}, n)$, and the final merge operation involves kn elements. The solution to this recurrence is $T_n(k, n) = nk \log k$. Note that n is a constant here across k arrays.

Exercise 3. DPV 2.23

Solution:

1. If there is a majority element in A, there must also be the majority element in either $A1$ or $A2$ or both.

   - If there is the same majority element in both, we return it as the majority of A
   - if $A1$ has a majority element E_0, we must check all elements in A against E_0 to determine if E_0 is indeed a majority element of A. $\mathcal{O}(n)$ runtime.
   - if $A2$ has a majority element E_1, we must check all elements in A against E_1 to determine if E_1 is indeed a majority element of A. $\mathcal{O}(n)$ runtime.

   At each stage, the worst case runtime occurs if both $A1$ and $A2$ have different majorities: $\mathcal{O}(2n) = \mathcal{O}(n)$
   Pseudocode

```
procedure majority($A[1, \cdots, n]$)
if $n == 1$:
 return $A[1]$
Let $A1$ and $A2$ be the first and second halves of $A$
$M1$ = majority($A1$)
$M2$ = majority($A2$)
if $M1$ is a majority element of $A$:
 return $M1$
if $M2$ is a majority element of $A$:
 return $M2$
return "no majoirty"
```

   Running time: $T(n) = 2T(n/2) + O(n) = O(n \log n)$

2. We pair up the elements of A arbitrarily, to get $n/2$ pairs. For each pair, if the two elements are different, we discard both of them else we keep just one of them.

   Pseudocode

```
procedure majority($A[1..n]$)
$x$ = prune($A$)
if $x$ is a majority element of $A$:
 return $x$
else:
 return "no majority"

procedure prune($S[1..n]$)
if $n == 1$:
 return $S[1]$
if $n$ is odd:
 if $S[n]$ is a majority element of $S$:
  return $S[n]$
 $n = n - 1$
$S'$ = [ ] (empty list)
for $i = 1$ to $n/2$:
 if $S[2i-1] = S[2i]$:
  add $S[2i]$ to $S'$
```

```
    return prune(S')
```

Justification: We'll show that each iteration of the **prune** procedure maintains the following invariant: if x is a majority element of S then it is also a majority element of S'. The rest then follows.
Suppose x is a majority element of S. In an iteration of **prune**, we break S into pairs. Suppose there are k pairs of Type One and l pairs of Type Two:

- Type One: the two elements are different. In this case, we discard both.
- Type Two: the elements are the same. In this case, we keep one of them.

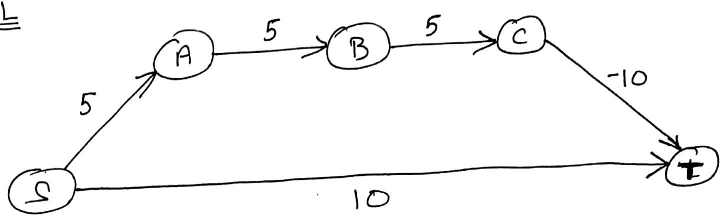
Since x constitutes at most half of the elements in the Type One pairs, x must be a majority element in the Type Two pairs. At the end of the iteration, what remains are l elements, one from each Type Two pair. Therefore x is the majority of these elements.

Running time. In each iteration of **prune**, the number of elements in S is reduced to $l \leq |S|/2$, and a linear amount of work is done. Therefore, the total time taken is $T(n) \leq T(n/2) + \mathcal{O}(n) = \mathcal{O}(n)$.

Exercise 4. DPV 4.8

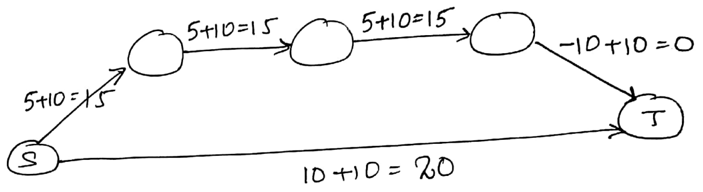
Solution: No, the algorithm suggested by Professor Lake for finding shortest path in a directed graph with negative edges is not correct. This can be seen in the example shown below. Note that Prof. Lake's algorithm penalizes paths with more number of edges more than paths with lesser number of edges. In the original graph, the shortest path is S,A,B,C,T whose cost/distance is 5. In the modified graph, the shortest path is S,T whose cost in the original graph is 10. Thus, we see that the shortest path in the original graph and the modified graph are not the same, making Professor Lake's algorithm incorrect.

ORIGINAL



Shortest path = SABCT
Shortest path distance = 5

MODIFIED :



Shortest path = ST
Shortest path distance = 20 (Modified)
= 10 (Original)

Exercise 5. DPV 4.18

Solution:

- **Algorithm:** This is a variation of Dijkstra's algorithm of Figure 4.8 (DPV). In the initialization loop, *best*$[s]$ is set to 0 and all other entries of *best* are set to ∞. In every iteration, *best* is also updated along with *distance*. The pseudocode below makes the changes more clear.

- **Pseudocode:**

  - Input:
    - Input $G = (V, E)$.
    - Positive edge lengths $\{l_e : e \in E\}$.
    - Source vertex $s \in V$.
  - Output: Array *best*$[u]$ with values set for all $u \in V$.

```
procedure find_best(G, l, s)
 for all $u \in V$:
  dist[u] = $\infty$
  best[u] = $\infty$
  prev[u] = nil
 dist[s] = 0
 H = Makequeue(V)
 // Using dist and best values as keys.
 // dist is the primary key and best is the secondary key.
 // (The secondary key is used to break tie in the primary keys)
 while H is not empty:
  u = deletemin(H)
  for all edges (u, v) $\in$ E:
  if dist(v) > dist(u) + l(u, v):
   dist(v) = dist(u) + l(u, v)
   best(v) = best(u) + 1
   decreasekey(H, v)
  else if dist(v) = dist(u)+l(u,v) and best[v] > best[u]+1:
   best[v] = best[u]+1
   decreasekey(H, v)
 return best
```

- **Proof of Correctness:** The proof of correctness of the above algorithm is along the same lines as that of Dijkstra's.

- **Time Complexity:** This has the same asymptotic running time as the original Dijkstras algorithm, as the additional operations in the loop take constant time.

Problem 1. DPV 2.5

Solution:

(a) $T(n) = 2T(\frac{n}{2}) + 1$.
Applying the Master theorem, we have $a = 2$, $b = 2$, $d = 0$, leading to $d < \log_b a$. Therefore, $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$.

(b) $T(n) = 5T(\frac{n}{4}) + n$.
Applying the Master theorem, we have $a = 5$, $b = 4$, $d = 1$, leading to $d < \log_b a$. Therefore, $T(n) = \Theta(n^{\log_4 5})$.

(c) $T(n) = 7T(\frac{n}{7}) + n$.
Applying the Master theorem, we have $a = 7$, $b = 7$, $d = 1$, leading to $d = \log_b a$. Therefore, $T(n) = \Theta(n^d \log n) = \Theta(n \log n)$.

(d) $T(n) = 9T(\frac{n}{3}) + n^2$.
Applying the Master theorem, we have $a = 9$, $b = 3$, $d = 2$, leading to $d = \log_b a$. Therefore, $T(n) = \Theta(n^d \log n) = \Theta(n^2 \log n)$.

(e) $T(n) = 8T(\frac{n}{2}) + n^3$.
Applying the Master theorem, we have $a = 8$, $b = 2$, $d = 3$, leading to $d = \log_b a$. Therefore, $T(n) = \Theta(n^d \log_2 n) = \Theta(n^3 \log n)$.

(f) $T(n) = 49T(\frac{n}{25}) + n^{\frac{3}{2}} \log n$
The size of the subproblem decreases dramatically (by 25 times). We guess that the complexity will be dominated by $n^{\frac{3}{2}} \log n$. Let's prove this:

Lower bound: $T(n) = 49T(\frac{n}{25}) + n^{\frac{3}{2}} \log n \geq n^{\frac{3}{2}} \log n$, thus, $T(n) = \Omega(n^{\frac{3}{2}} \log n)$.

Upper bound: Let's show that $T(n) \leq cn^{\frac{3}{2}} \log n$ for some c and find possible values for c using constructive induction. We will use strong induction and assume that $T(m) \leq cm^{\frac{3}{2}} \log m$ for all $m < n$. Thus, we have $T(n) = 49T(\frac{n}{25}) + n^{\frac{3}{2}} \log n \leq 49c(\frac{n}{25})^{\frac{3}{2}} \log(\frac{n}{25}) + n^{\frac{3}{2}} \log n = \frac{49}{125}cn^{\frac{3}{2}} \log(\frac{n}{25}) + n^{\frac{3}{2}} \log n \leq (\frac{49}{125}c + 1)n^{\frac{3}{2}} \log n \leq cn^{\frac{3}{2}} \log n$. Thus, we need the following inequality to be satisfied: $c \geq \frac{125}{76}$. For any such values c, the statement will be true. We conclude that $T(n) = \mathcal{O}(n^{\frac{3}{2}} \log n)$. Therefore, we have $T(n) = \Theta(n^{\frac{3}{2}} \log n)$.

(g) $T(n) = T(n-1) + 2$.

$$T(n) = T(n-2) + 2 + 2 = \cdots = T(n-i) + 2i = \cdots = T(0) + 2n = \Theta(n)$$

(h) $T(n) = T(n-1) + n^c$.

$$T(n) = T(n-2) + n^c + (n-1)^c = \cdots = T(n-i) + \sum_{j=0}^{j=i-1} (n-j)^c = \cdots = T(0) + \sum_{j=0}^{j=n-1} (n-j)^c = \Theta(n^{c+1})$$

(i) $T(n) = T(n-1) + c^n$.

$$T(n) = T(n-2) + c^n + c^{n-1} = \cdots = T(n-i) + \sum_{j=0}^{j=i-1} c^{n-j} = \cdots = T(0) + \sum_{j=0}^{j=n-1} c^{n-j} = \Theta(c^n)$$

(j) $T(n) = 2T(n-1) + 1$.

$$T(n) = 2(2T(n-2)+1)+1 = 4T(n-2)+3 = \cdots = 2^i T(n-i) + 2^i - 1 = \cdots = 2^n T(0) + 2^n - 1 = \Theta(2^n)$$

(k) $T(n) = T(\sqrt{n}) + 1$.

Let $2^{2^k} = n$ (For $\mathcal{O}$ bound you can assume $k = \lceil \log(\log n)) \rceil$ and for Ω bound, you can assume $k = \lfloor \log(\log n)) \rfloor$. Now let $T(n) = T(2^{2^k}) = S(k)$. Therefore, $T(\sqrt{n}) = T(2^{\frac{2^k}{2}}) = S(k-1)$. The new relation we have is, $S(k) = S(k-1) + 1$. From part (g), we know that $S(k) = \Theta(k) = \Theta(\log(\log n)) = T(n)$.

Problem 2. 2.22

Solution:

- **Algorithm:** We are searching for the k^{th} smallest element s_k in the union of the subarrays $a[1, \cdots, n]$ and $b[1, \cdots, m]$. Because we are looking at the k^{th} smallest element, we can restrict our attention to the arrays $a[1, \cdots, k]$ and $b[1, \cdots, k]$. If $k > m$ or $k > n$, we can take all the elements with index larger than the array boundary to have infinite value. Our algorithm starts off by comparing $a[\lfloor \frac{k}{2} \rfloor]$ and $b[\lceil \frac{k}{2} \rceil]$:
  - If $a[\lfloor \frac{k}{2} \rfloor] > b[\lceil \frac{k}{2} \rceil]$, then s_k will be the $\lfloor \frac{k}{2} \rfloor^{th}$ smallest element of the union of the subarrays $a[1, \cdots, \lfloor \frac{k}{2} \rfloor]$ and $b[\lceil \frac{k}{2} \rceil + 1, \cdots, k]$.
  - If $a[\lfloor \frac{k}{2} \rfloor] < b[\lceil \frac{k}{2} \rceil]$, then s_k will be the $\lceil \frac{k}{2} \rceil^{th}$ smallest element of the union of the subarrays $a[\lceil \frac{k}{2} \rceil + 1, \cdots, k]$ and $b[1, \cdots, \lfloor \frac{k}{2} \rfloor]$.
  - The base case of the recursion is $a[\lfloor \frac{k}{2} \rfloor] = b[\lceil \frac{k}{2} \rceil]$ and we return $s_k = a[\lfloor \frac{k}{2} \rfloor] = b[\lceil \frac{k}{2} \rceil]$.
- **Pseudocode:**
  - Input:
    - First array a.
    - Second array b.
    - Integer k.
  - Output: The k^{th} smallest element in the union of two lists a and b.

```
procedure smallest(a, b, k)
 a = restrict(a, k)
 b = restrict(b, k)

 fl = floor(k / 2)
 cl = ceil(k / 2)
 if a[fl - 1] > b[cl - 1]:
  return smallest(a[0..fl - 1], b[cl..k - 1], fl)
 else if a[fl - 1] < b[cl - 1]:
  return smallest(a[fl..k - 1], b[0..cl - 1], cl)
 else
  return a[cl - 1]

// makes k the length of the array a by either:
// - restricting it if length > k, or
// - adding INFINITY values at the end

procedure restrict(a, k)
 n = size(a)
 if k <= n - 1:
  a = a[0...k-1]
 else:
  a = a + [INFINITY] * (k - n)
 return a
```

- **Proof of Correctness:** Our algorithm starts off by comparing $a[\lfloor \frac{k}{2} \rfloor]$ and $b[\lceil \frac{k}{2} \rceil]$:
  - Suppose $a[\lfloor \frac{k}{2} \rfloor] > b[\lceil \frac{k}{2} \rceil]$. Then:

      * In the union of a and b there can be at most $k-2$ elements smaller than $b[\lceil\frac{k}{2}\rceil]$, i.e. $a[1,\cdots,\lfloor\frac{k}{2}\rfloor-1]$ and $b[1,\cdots,\lceil\frac{k}{2}\rceil-1]$ and we must necessarily have $s_k > b[\lceil\frac{k}{2}\rceil]$.
      * Similarly, all elements $a[1,\cdots,\lfloor\frac{k}{2}\rfloor]$ and $b[1,\cdots,\lceil\frac{k}{2}\rceil]$ will be smaller than $a[\lfloor\frac{k}{2}\rfloor+1]$. But these are k elements, so we must have $s_k < a[\lfloor\frac{k}{2}\rfloor+1]$.

    This shows that s_k must be contained in the union of the subarrays $a[1,\cdots,\lfloor\frac{k}{2}\rfloor]$ and $b[\lceil\frac{k}{2}\rceil+1,\cdots,k]$.
    In particular, as we discarded $\lceil\frac{k}{2}\rceil$ elements smaller than s_k, then s_k will be the $\lfloor\frac{k}{2}\rfloor^{th}$ smallest element in this union.
    We can then find s_k by recursing on this smaller problem.

  – The case $a[\lfloor\frac{k}{2}\rfloor] < b[\lceil\frac{k}{2}\rceil]$ is symmetric.
  – The last case, which is also the base case of the recursion, is $a[\lfloor\frac{k}{2}\rfloor] = b[\lceil\frac{k}{2}\rceil]$, for which we have $s_k = a[\lfloor\frac{k}{2}\rfloor] = b[\lceil\frac{k}{2}\rceil]$.

- **Time Complexity:** Our algorithm is characterized by the recurrence:

$$T(k) = T(\frac{k}{2}) + \mathcal{O}(1)$$

  Using the Master Theorem with $a = 1$, $b = 2$ and $d = 0$, as $\log_b a = 0 = d$ we get $T(k) = \mathcal{O}(\log k) = \mathcal{O}(\log n + \log m)$.

Problem 3

(a) DPV 2.12

Solution: Each function call prints one line and calls the same function on input of half the size, so the number of printed lines is $P(n) = 2P(\frac{n}{2}) + 1$. Applying the Master theorem, we have $a = 2, b = 2, d = 0$, leading to $d < \log_b a$. Therefore, $P(n) = \Theta(n^{\log_b a}) = \Theta(n)$.

(b) DPV 2.14

Solution:

- **Algorithm:** Sort the array. Then scan the array from left to right and copy the elements into a new array eliminating the duplicates.
- **Pseudocode:**
  - Input: An array with n elements.
  - Output: An array with the same elements as the input array but without any duplicates.

```
procedure eliminate_duplicates(A[1, ..., n])
B = []
A = sort(A)
B.append(A[1])
prev = A[1]
for i in 2, ..., $n$
 if prev = A[i]
  continue
 B.append(A[i])
 prev = A[i]
return B
```

- **Proof of Correctness:**

  Claim 1: The algorithm removes ALL the duplicates.
  Proof: We note that once we sort A, all the duplicates come together in the sorted array. Thus when we see the first occurrence of the element and update *prev* to be that value, it makes it impossible to append any following occurrences of that element into B.

  Claim 2: The algorithm removes ONLY the duplicates.
  Proof: The algorithm scans each element of the sorted array and chooses to skip appending that element to the output only when the value of that element is equal the previous element's value. Since, for the first occurrence of an element, this condition will always be false, we will always append the first occurrence of any element in the sorted array to B.

  Claims 1 and 2 together prove the correctness of our algorithm.

- **Time Complexity:** The algorithm performs sorting ($\mathcal{O}(n \log n)$) followed by a single scan from left to right ($\mathcal{O}(n)$). Therefore, the total complexity is $\mathcal{O}(n \log n)$).

(c) DPV 2.16

Solution:

- **Algorithm:**

  1. Find n.
     - To find n, query elements $A[1], A[2], \cdots, A[2^i], \ldots$ until you find the first element $A[2^k]$ such that $A[2^k] = \infty$. Then, $2^{k-1} \leq n < 2^k$.
     - We can then perform binary search on $A[2^{k-1}, \ldots, 2^k]$ to find the last non-infinite element of A.
  2. Once we have found n, we can perform binary search on $A[1, \ldots, n]$ to check if given element x exists in A or not.

- **Pseudocode:**

  - Input: Array $A[1, \ldots, n, \ldots]$ with infinite size. $A[i] = \infty$ for $i > n$.
  - Output: Index of the element x in array if it exists, -1 if it doesn't.

```
find_n(A[1,...,n,...])
 k := 0
 while $A[2^k]$ is not equal to $\infty$
  k := k+1
 low := $2^{k-1}$
 high := $2^k$
 while high - low > 2
  mid := (high + low)/2
  if A[mid] is equal to $\infty$
   high := mid
  else
   low := mid
 return low
```

- **Proof of Correctness:**

  Claim 1: The algorithm correctly finds the two consecutive powers of two between which n lies.
  Proof: This is clear because the algorithm queries every power of 2 in increasing order and since n is finite, there will be a power of 2 greater than n, *high*, such that $A[high] = \infty$ and a power of 2 smaller than n, *low*, such that $A[low]$ is a finite number and n will lie between these two numbers.

  Claim 2: Throughout the algorithm, $A[high] = \infty$.
  Proof: We start off with $A[high] = \infty$ (remember that *high* was the first power of two such that $A[high] = \infty$). In the algorithm, we only update *high* to *mid* when $A[mid] = \infty$. Therefore, $A[high]$ remains ∞ even after the update.

  Claim 3: Throughout the algorithm, $A[low]$ is a finite number.
  Proof: We start off with $A[low]$ being a finite number (remember that *low* was the biggest power of two such that $A[low] \neq \infty$). In the algorithm, we only update *low* to *mid* when $A[mid]$ is a finite number. Therefore, $A[low]$ remains a finite number even after the update.

Claim 4: n always remains between *low* and *high*.
Proof: This follows from Claims 2 and 3.

Claim 5: Base Case: If $high - low = 1$. $n = low$.
Proof: This follows from Claims 2, 3 and 4.

Claims 1 to 5 together prove the correctness of our algorithm.

- **Time Complexity:**
  - Finding *high* takes $\mathcal{O}(\log n)$ time.
  - Finding n takes $\mathcal{O}(\log n)$ time.
  - Finding if x exists takes $\mathcal{O}(\log n)$ time.

Therefore, the total runtime of the algorithm is $\mathcal{O}(\log n)$.

Problem 4. DPV 2.27

Solution:

(a)

$$M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}^2 = \begin{pmatrix} a^2 + bc & b(a+d) \\ c(a+d) & d^2 + cb \end{pmatrix}$$

Therefore, to compute the above matrix, the following 5 multiplications are sufficient, $a^2, d^2, b(a+d), c(a+d), bc$.

(b) We do get 5 subproblems but they are not of the same type as the original problem. Note that we started with a squaring problem for a matrix of size nxn and three of the 5 subproblems now involve multiplying $\frac{n}{2}$x$\frac{n}{2}$ matrices. Hence the recurrence $T(n) = 5T(\frac{n}{2}) + O(n^2)$ does not make sense.

(c) i. Note that, $AB + BA = (A+B)(A+B) - AA - BB$.

- AA can be computed in S(n) from A.
- BB can be computed in S(n) from B.
- A+B can be computed in $\mathcal{O}(n^2)$ from A and B.
- (A+B)(A+B) can be computed in S(n) from (A+B).
- (A+B)(A+B) - AA - BB can be computed in $\mathcal{O}(n^2)$ from (A+B)(A+B), AA and BB.

Therefore, total time is 3S(n)+$\mathcal{O}(n^2)$

ii. $AB + BA = \left(\begin{smallmatrix} 0 & XY \\ 0 & 0 \end{smallmatrix}\right)$

iii. We see from (ii) that XY is the upper-right submatrix of $AB + BA$. Therefore, it can be computed in $\mathcal{O}((2n)^2) = \mathcal{O}(n^2)$ from $AB + BA$. Also, in (i), we saw that $AB + BA$ can be computed in $3S(2n) + \mathcal{O}((2n)^2) = \mathcal{O}((2n)^c) + \mathcal{O}((2n)^2) = \mathcal{O}(n^c) + \mathcal{O}(n^2)$. Therefore, total runtime $T(n) = \mathcal{O}(n^2) + \mathcal{O}(n^c) + \mathcal{O}(n^2) = \mathcal{O}(n^c)$.

Problem 5. DPV 4.5

Solution:

- **Algorithm:** We will use BFS for this algorithm to track the number of paths to node v. We store a paths counter for any node v at depth l in the BFS tree by adding the number of paths to the node v from edges (u, v) where u is at depth $l-1$. If u is not at depth $l-1$ we do not add its paths counter to the total stored for v.
- **Pseudocode:**
  - Input
    - Undirected graph $G(V, E)$.
    - Nodes $u, v \in V$.
  - Output: The number of distinct shortest paths from u to v.

```
procedure count_paths(G, u, v)
 for all x in V:
  dist[x] = Infinity   num_paths[x] = 0

 dist[u] = 0
 num_paths[u] = 1
 Q = [u]
 While Q is not empty:
  x = eject(Q)
  for all edges (x, y) in E:
   if dist[y] = dist[x] + 1:
    num_paths[y] = num_paths[y] + num_paths[x]
   if dist[y] = Infinity
    inject(Q, y)
    dist[y] = dist[x] + 1
    num_paths[y] = num_paths[x]
 return num_paths[v]
end
```

- **Proof of Correctness:** Consider the BFS search tree constructed for G. We are counting only paths that go through the nodes connected to u at one level above u in the tree. Since we disregard nodes that connect to u but are at the same level or lower in the BFS tree, we only counts shortest paths. Therefore, when the algorithm terminates, we will have the number of shortest paths to v.
- **Time Complexity:** We slightly modify BFS to store a paths counter as we proceed through the graph G. This only requires a constant amount of time per edge and so the time complexity is the same as BFS, $\mathcal{O}(|V| + |E|)$.

Problem 6. DPV 4.14

Solution:

- **Algorithm:** To find all the shortest paths from u to v for all $u, v \in V$, through some vertex v_0, it is sufficient to find the shortest path distances from every node u to v_0 and from v_0 to every node v. The shortest distances from v_0 to every node $v \in V$ can be found by running Dijkstra's algorithm on G. The shortest distances from every node $v \in V - \{v_0\}$ to v_0 can be found by running Dijkstra's algorithm on the reversed graph G^R starting from v_0.
  We can perform this through the following algorithm:

  1. Reverse the graph G to obtain reverse graph G^R, retaining all edge weights in the process.
  2. Execute Dijkstra's algorithm on G^R with source vertex v_0 to obtain the shortest path distances from v_0 to all other vertices. This set of distances is the set of shortest path distances from all vertices to v_0 in G.
  3. Execute Dijkstra's algorithm on G with source vertex v_0 to obtain the shortest path distances from v_0 to all other vertices.
  4. For each pair of vertices u, v, the distance from u to v through v_0 is the sum of the distance from u to v_0 as obtained in Step 2, and the distance from v_0 to v as obtained in Step 3.

- **Pseudocode:**

```
Reverse G to obtain GR

dist_to_v0[.]   = dijkstra($G^R$; $v_0$)
dist_from_v0[.]  = dijkstra(G; $v_0$)

for all $u \in V$ :
 for all $v \in V$ :
  dist(u,v) = dist_to_v0[u] + dist_from_v0[v]
```

- **Proof of Correctness:**
  Proof by construction:
  In Step 2 and Step 3, Dijkstra's algorithm correctly finds the shortest distance from v_0 to V in G^R and G respectively. Since the shortest path between any pair of nodes is formed by joining two subpaths of shortest lengths, it follows that the paths found by this algorithm are the shortest.

- **Time Complexity:** Reversing the graph G can be performed in $\mathcal{O}(|V| + |E|)$ time. The two executions of Dijkstra's algorithm take $\mathcal{O}((|V| + |E|) \log |V|)$ time. The computation of shortest path between all pairs of nodes through v_0 takes a further $\mathcal{O}(|V|^2)$ time. Therefore, the overall time complexity of the algorithm is $\mathcal{O}(|V|^2 + |E| \log |V|)$.

Problem 7. DPV 4.19

Solution:

- **Algorithm:** Our algorithm solves the node cost problem by reducing it to a problem that only contains edge lengths and no longer involves vertex costs. First, we create a node u' for each node u and replace each directed edges $e = (u, v)$ by new edges $e' = (u', v)$. Then, an edge $e_c = (u, u')$ is added with a length equal to the cost of vertex u. We run Dijkstra's on this new graph and return the shortest path omitting any added edges e_c.
- **Pseudocode:**
  - Input:
    - A directed graph $G = (V, E)$ having positive edge lengths and positive vertex costs.
    - A starting vertex s in V.
  - Output: The array $cost[]$ of costs.

```
procedure shortest_paths_with_vertex_costs(G, s)
 for each v in V:
  add new vertex v'
  add edge (v, v') with length v.cost
  move all edges (v, u) to (v', u)
 end
 run Dijkstras on (G)
 remove cost values for all vertex v' added above
 return costs array
end
```

- **Proof of Correctness:** Consider the path through vertex u in the original graph G. Notice that this path can be converted to a path of the same total cost by replacing the cost of vertex u with the traversal of the edge (u, u'). Conversely, consider the path through vertex u in the modified graph. Notice that this path can be converted to a path of the same total cost by replacing the edge (u, u') with the cost of vertex u. So, we can reduce the shortest paths with vertex costs problem to an instance of the shortest paths problem.
- **Time Complexity:** We step through each vertex $v \in V$ once to add new vertex v' to G. This requires $O(|V|)$ steps since we can perform all needed loop steps in constant time (all edges can be moved via two pointer manipulations given an adjacency list format). Dijkstra's is unmodified and continues to run in $\mathcal{O}(|V|)^2$ time. Our overall running time is therefore the same as Dijkstra's.