Nathan Bickel
CSCE 350: Section H01
Professor: Dr. Narayanan
March 21, 2023

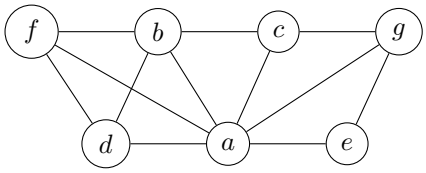
CSCE 350 Homework 4

Problem 1 Consider the graph in Figure 1.

(a) Write down the adjacency matrix and adjacency lists specifying this graph. (Assume that the matrix rows and columns and vertices in the adjacency lists follow in the alphabetical order of the vertex labels.) (5 points)

(b) Starting at vertex a and resolving ties by the vertex alphabetical order, traverse the graph by depth-first search and construct the corresponding depth-first search forest including the tree edges and the other types of edges. You should use solid edges to represent tree edges and dashed edges to represent other edges. Give the order in which the vertices were reached for the first time (pushed onto the traversal stack) and the order in which the vertices became dead ends (popped off the stack). (5 points)

(c) Starting the traversal at vertex a and resolve ties by the vertex alphabetical order, traverse the graph by breadth-first search and construct the corresponding breadth-first search tree. You should use solid edges to represent tree edges and dashed edges to represent other edges. Give the order in which the vertices were reached for the first time (enqueue) and the order in which the vertices became dead ends (dequeue). (5 points)



Solution.

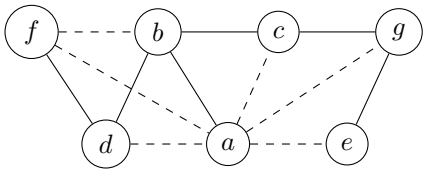
(a) We write the adjacent matrix

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

We can also write the adjacency linked list for A:

$$\{a : [b, c, d, e, f, g], b : [a, c, d, f], c : [a, b, g], d : [a, b, f], e : [a, g], f : [a, b, d], g : [a, c, e]\}.$$

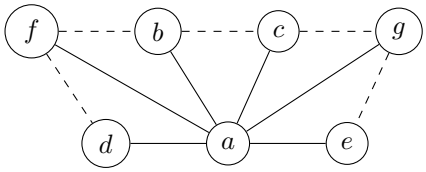
(b) We draw the depth-first-search forest:



Order Reached: a, b, c, g, e, d, f

Order of Dead Ends: e, g, c, f, d, b, a

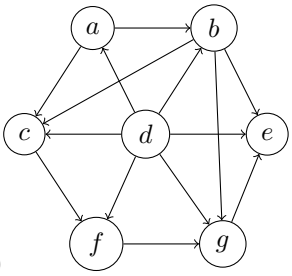
(c) We draw the breadth-first-search forest:



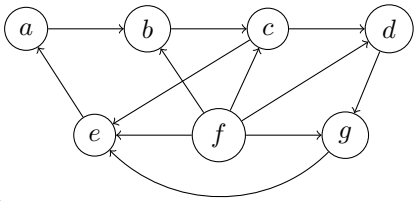
Order Reached: a, b, c, d, e, f, g

Order of Dead Ends: b, c, d, e, f, g, a

Problem 2 Apply the DFS-based algorithm to solve the topological sorting problem for the following digraphs in Figure 2. You should also give the order of the vertices visited and the DFS forest including the tree edges and the other types of edges. (10 points)



(a)

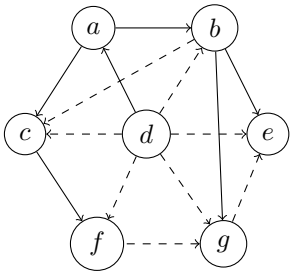


(b)

(a) We start at vertex d and apply the DFS algorithm.

Order Reached: d, a, b, e, g, c, f

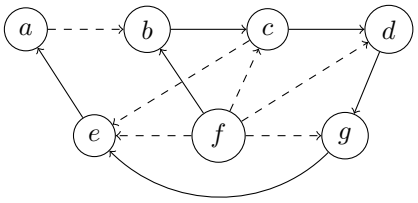
Order of Dead Ends: e, g, b, f, c, a, d



(b) We start at vertex f and apply the DFS algorithm.

Order Reached: f, b, c, d, g, e, a

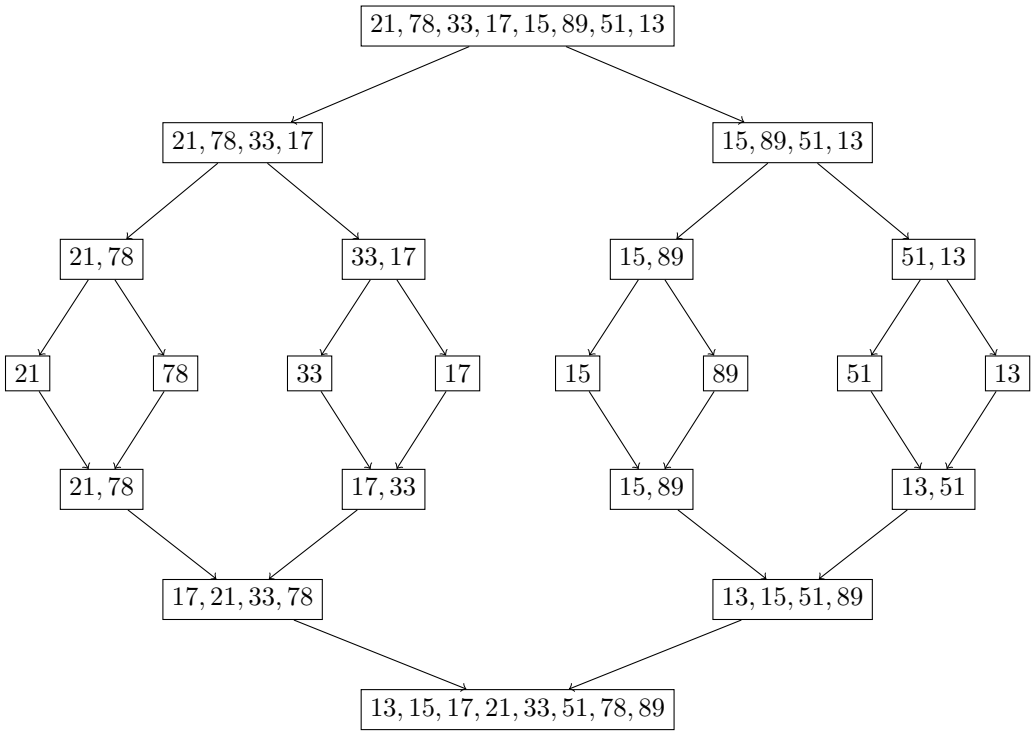
Order of Dead Ends: a, e, g, d, c, b, f



Problem 3 Apply mergesort to sort a list of numbers $21, 78, 33, 17, 15, 89, 51, 13$ in a nondecreasing order. Show the mergesort operation following the example in the textbook. How many comparisons do you need for sorting this list? (10 points)

Solution.

As in the textbook, we show the dividing and merging:



When merging two arrays B and C into array A, we choose the smaller of $B[0]$ and $C[0]$, remove it and append it to array A, continue until one of the lists is empty, and add the non-empty list to A. We have three merging levels:

1. In the first level, 4 comparisons are made (comparing the 4 pairs of numbers).
2. 3 comparisons are made to merge $21, 78$ and $17, 33$. 3 are also made to merge $15, 89$ and $13, 51$.
3. 7 comparisons are made to merge $17, 21, 33, 78$ and $13, 15, 51, 89$.

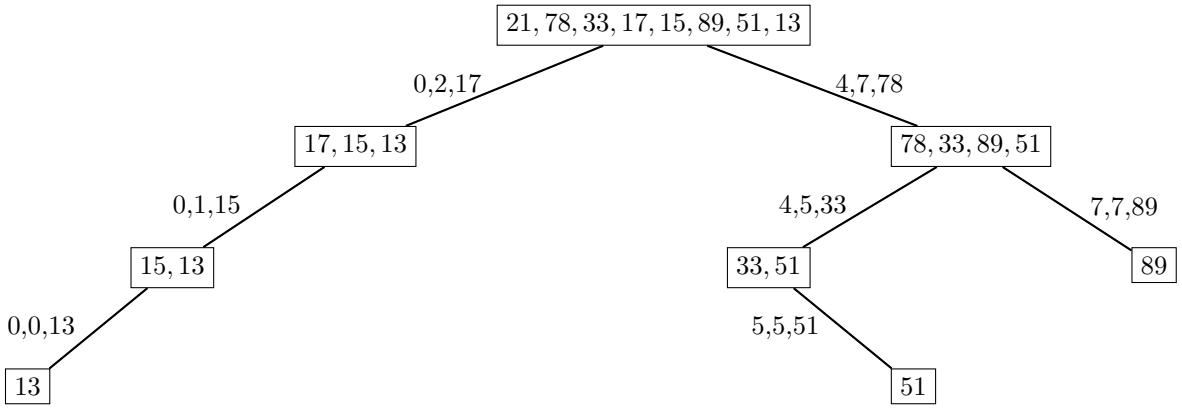
So in total, we make $4 + 3 + 3 + 7 = 17$ comparisons.

Problem 4 Apply quicksort to sort a list of numbers $21, 78, 33, 17, 15, 89, 51, 13$ in a nondecreasing order. You must show the step-by-step quicksort operations following the example in the textbook. A tree of

recursive calls with l, r, and pivot positions is required. How many comparisons do you need for sorting this list? (10 points)

Solution.

We begin with $l, r, p = 0, 7, 21$, and list the recursive calls in order l, r, p (in each box, we have the sub-array, and the position splits to divide the arrays in between):



So we end with the list $13, 15, 17, 21, 33, 51, 78, 89$. In each sublist, we compare each element to the pivot, so there is one fewer comparison made than items in the list. Counting this in each sublist in the tree, we make $(8-1)+(3-1)+(4-1)+(2-1)+(2-1)+(1-1)+(1-1)+(1-1) = 14$ comparisons.

Problem 5 Divide-and-conquer

(a) Write a pseudocode for a divide-and-conquer algorithm for the exponentiation problem of computing a^n, where $a > 0$ and n is a positive integer. (10 points)

(b) Set up and solve a recurrence relation for the number of multiplications made by this algorithm. (5 points)

(c) How does this algorithm compare with the brute-force algorithm for this problem? (5 points)

(d) How does this algorithm compare with the decrease-and-conquer algorithm for this problem? (5 points) (**Hint**: How would you compute a^8 by solving two exponentiation problems of size 4? How about a^9?)

Solution.

(a) We show the divide-and-conquer algorithm below:

(b) We perform one multiplication if $2 \mid n$ and two multiplications if $2 \not| n$. Let $M(n)$ be the number of multiplications made by this algorithm. Then, we have

$$M(n) = \begin{cases} M\left(\frac{n}{2}\right) + 1 & \text{if } 2|n \\ M\left(\frac{n-1}{2}\right) + 2 & \text{otherwise} \end{cases} = M\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 + n \bmod 2; M(1) = 0.$$

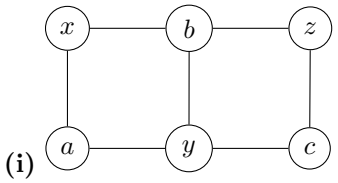
Let $d = 0$. Then $1 = 2^0 = 2^d$, so by the Master theorem, this algorithm is in $\Theta(n^0 \log n) = \Theta(\log n)$.

(c) This algorithm is better than the brute-force algorithm, which simply multiplies a by itself $n-1$ times, which is in $\Theta(n)$.

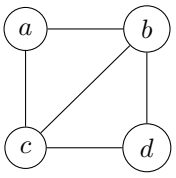
(d) A decrease-and-conquer algorithm for this problem is to fix some $k \in \mathbb{Z}$, decrease n by k, calculate a^k, and recursively solve $a^k a^{n-k}$. This requires $\left\lfloor \frac{n}{k} \right\rfloor + k$ multiplications, so this is in $\Theta(n)$ and our divide-and-conquer algorithm is better.

```
// Input: a ∈ ℝ⁺, n ∈ ℤ⁺
// Output: aⁿ
function POWER(a, n)
    if n = 1 then
        return a
    else if 2 | n then
        // n/2 is an integer since n is even
        temp ← POWER(a, n/2)
        // (a^(n/2))(a^(n/2)) = a^(n/2 + n/2) = aⁿ
        return temp × temp
    else
        // (n−1)/2 is an integer since n is odd
        temp ← POWER(a, (n−1)/2)
        // (a^((n−1)/2))(a^((n−1)/2))(a) = a^((n−1)/2 + (n−1)/2 + 1) = aⁿ
        return temp × temp × a
    end if
end function
```

Problem 6 A graph is said to be bipartite if all its vertices can be partitioned into two disjoint subsets X and Y so that every edge connects a vertex in X with a vertex in Y. (One can also say that a graph is bipartite if its vertices can be colored in two colors so that every edge has its vertices colored in different colors; such graphs are also called 2-colorable. For example, graph (i) is bipartite while graph (ii) is not.)



(i)



(ii)

(a) Design a DFS-based algorithm for checking whether a graph is bipartite. Write the pseudo-code and traversal forest. (10 points)

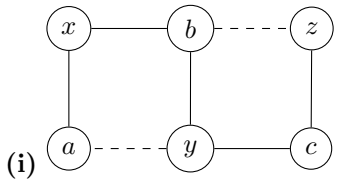
(b) Design a BFS-based algorithm for checking whether a graph is bipartite. Write the pseudo-code and traversal forest. (10 points)

Solution.

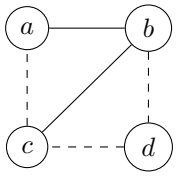
(a) We create an algorithm (below) to check if a connected component of a graph is bipartite. To check if a graph with multiple components is connected, one can run it starting at a vertex in each component.

We then apply the algorithm to the two graphs, starting from a and showing the traversal trees (we break ties by alphabetical order):

```
X ← ∅
Y ← ∅
// Input: A graph G and starting vertex v ∈ V(G)
// Output: True if the component of G with v is bipartite and false if not
function DFSCheckBipartite(G, v)
    vPlaceCandidate ← Null // Will be assigned to X or Y
    // Find where v can't go: if it's in one set, put it in the other, and if it's both sets, return false
    for v' ∈ N(v) do
        // Check all neighbors to see restrictions on placement of v
        if v' ∈ X then
            if vPlaceCandidate = Null then
                vPlaceCandidate ← Y
            else if vPlaceCandidate = X then
                // v has neighbors in X and Y, so G can't be bipartite
                return false
            end if
        else if v' ∈ Y then
            if vPlaceCandidate = Null then
                vPlaceCandidate ← X
            else if vPlaceCandidate = Y then
                // v has neighbors in X and Y, so G can't be bipartite
                return false
            end if
        end if
    end for
    if vPlaceCandidate = X then
        X ← X + {v}
    else if vPlaceCandidate = Y then
        Y ← Y + {v}
    else
        We didn't find any restrictions, so v must be the starting vertex: arbitrarily place it in X
        X ← {v}
    end if
    // Recursively run the algorithm until every vertex has been covered
    for v' ∈ N(v) do
        if v' ∉ X ∪ Y then
            if !DFSCheckBipartite(G, v') then
                // We can't create a bipartition
                return false
            end if
        end if
    end for
    return true // We made a bipartition!
end function
```



(i)

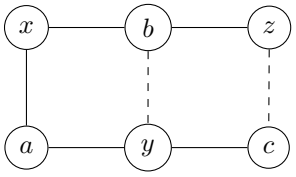


(ii)

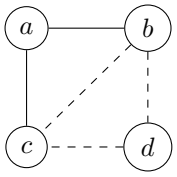
In (i), we obtain the bipartition $X = \{a, b, c\}$ and $Y = \{x, y, z\}$ with no issues, so this graph is bipartite. However, in (ii), our algorithm attempts to put a and c in the same independent set, but $ac \in E(G)$, so it returns false.

(b) Now, we create an algorithm (below) that uses a breadth-first search approach. Again, this will check if a connected component of a graph is bipartite, and the algorithm should be run on a vertex in each component if there are multiple. To start, choose some $v \in V(G)$ and run BFSCheckBipartite$(G, X, \{v\})$.

We then apply the algorithm to the two graphs, starting from a and showing the traversal trees (we break ties by alphabetical order):



(i)



(ii)

In (i), we obtain the bipartition $X = \{a, b, c\}$ and $Y = \{x, y, z\}$ with no issues, so this graph is bipartite. However, in (ii), our algorithm finds that $E(N(a)) = E(\{b, c\}) = \{bc\} \neq \emptyset$, so it returns false.

Problem 7 Consider ternary search—the following algorithm for searching in a sorted array $A[0, \ldots, n-1]$. If $n = 1$, simply compare the search key K with the single element of the array; otherwise, search recursively by comparing K with $A[\lfloor n/3 \rfloor]$, and if K is larger, compare it with $A[\lfloor 2n/3 \rfloor]$ to determine in which third of the array to continue the search.

(a) Set up a recurrence for the number of key comparisons in the worst case. You may assume that $n = 3^k$. (5 points)

(b) Solve the recurrence for $n = 3^k$. (5 points)

Solution.

(a) The worst case is that the key is not in the list. At each step, we compare it to the $(n/3)^{\text{th}}$ and $(2n/3)^{\text{th}}$ elements, so we make two comparisons unless there is only one element. Let $C(n)$ be the number of comparisons made with n elements. Then, we have

$$C(n) = C\left(\frac{n}{3}\right) + 2; C(1) = 1.$$

(b) We observe that $C(3^0) = 1, C(3^1) = 3, C(3^2) = 5, C(3^3) = 7, \ldots$ so using forward substitution we have $C(3^k) = 2k + 1$ for all $k \in \mathbb{N}$. Let $k = \log_3 n$, which will be an integer when n is a power of 3. So

$$C\left(3^{\log_3 n}\right) = C(n) = 2\log_3 n + 1.$$

```
X ← ∅
Y ← ∅
// Input: A graph G, set Z that is either X or Y, and set of vertices S ⊆ V(G) with no edges in B. S should be a subset of a connected component of G.
// Output: True if the component of G with S is bipartite and false if not
function BFSCheckBipartite(G, Z, S)
    if S = ∅ then
        return true // Vacuously works
    else if X = ∅ then
        X ← S
    else if Y = S then
        Y ← S
    else
        if Z = X then
            X ← X + S
            Z' ← Y
        else if Z = Y then
            Y ← Y + S
            Z' ← X
        end if
        // Check if N(S) can go in Z'
        for v ∈ N(S) do
            if N(v) ∩ Z' ≠ ∅ or E(N(v)) ≠ ∅ then
                // v has (or will have) edges in X and Y, so G can't be bipartite
                return false
            end if
        end for
    end if
    // Try to keep going until we've covered all the vertices
    while X ∪ Y ≠ V(G) do
        // Check if we can keep going with the vertices we haven't visited yet
        if !BFSCheckBipartite(G, Z', N(S) − Z') then
            return false // We got a collision with coloring
        end if
    end while
    return true // We found a bipartition!
end function
```

Problem 8 Design a divide-and-conquer algorithm for computing the number of levels in a binary tree. (In particular, the algorithm must return 0 and 1 for the empty and single-node trees, respectively.) What is the time efficiency class of your algorithm? (10 points)

Solution.

We design a divide-and-conquer algorithm:

```
// Input: Binary tree T rooted at v
// Output: Height of tree
// Note: If a tree is empty and thus has no root, set v = Null.
function BinTreeHeight(T, v)
    if v = Null then
        return 0
    else
        // We assume v has at most two children, leftChild and rightChild.
        // If one or both of these do not exist, the attribute should be set to Null.
        return 1 + max{BinTreeHeight(T, v.leftChild), BinTreeHeight(T, v.rightChild)}
    end if
end function
```

This is in $O(n)$, where n is the number of vertices, since in the worst case T is a path with endpoint v and we visit every vertex.

Problem 9 Revisiting Mergesort (10 points):

(a) Set up a recurrence relation for the number of key comparisons made by mergesort in the worst case. You may assume that $n = 2^k$.

(b) Solve the recurrence relation for the number of key comparisons made by mergesort on best-case inputs and solve it for $n = 2^k$.

Solution.

(a) In a worst case during the merging step, we have two lists $a_1, a_2, \ldots, a_{n/2}$ and $b_1, b_2, \ldots, b_{n/2}$ such that the order of choosing elements is $a_1, b_1, a_2, b_2, \ldots, a_{n/2}, b_{n/2}$. So we must make $n - 1$ comparisons, and we include $b_{n/2}$ at the end since the first list is first empty during this step. Let $C(n)$ be the number of comparisons. Then, we have

$$C(n) = 2C\left(\frac{n}{2}\right) + n - 1; C(1) = 0.$$

(b) A best case during the merging step is that we have two lists such that the last element of the first list is less than the first element of the second list. This way, we will only have to make $\frac{n}{2}$ comparisons, so we have

$$C(n) = 2C\left(\frac{n}{2}\right) + \frac{n}{2}; C(1) = 0.$$

Let $d = 1$. Then, $2 = 2^1 = 2^d$, so by the Master Theorem this time complexity is in $\Theta(n^d \log n) = \Theta(n \log n)$. Using forward substitution, we obtain $C(n) = n \log_4 n$.