multicol

Summary of COMP523 Advanced Algorithm

May 4, 2023

Chapter 1

Symmetry Notation

1.1 Asymptotic Notation

Asymptotic notation is a way of describing the limiting behavior of a function when the argument tends towards a particular value or infinity. In computer science, asymptotic notation is frequently used to describe the running time or space usage of an algorithm.

- O-notation: $f(n) = O(g(n))$ if there exist constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.
- Ω-notation: $f(n) = \Omega(g(n))$ if there exist constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.
- Θ-notation: $f(n) = \Theta(g(n))$ if there exist constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.
- o-notation: $f(n) = o(g(n))$ if for any constant $c > 0$, there exists a constant n_0 such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$.
- ω-notation: $f(n) = \omega(g(n))$ if for any constant $c > 0$, there exists a constant n_0 such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$.

1.2 Comparing Functions

1.2.1 Transitivity

- $f(n) = O(g(n))$ and $g(n) = O(h(n))$ implies $f(n) = O(h(n))$.
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ implies $f(n) = \Omega(h(n))$.
- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ implies $f(n) = \Theta(h(n))$.

For example, $n^2 = O(n^3)$ and $n^3 = O(n^4)$ implies $n^2 = O(n^4)$.

1.2.2 Reflexivity

- $f(n) = O(f(n))$.
- $f(n) = \Omega(f(n))$.
- $f(n) = \Theta(f(n))$.

For example, $n^2 = O(n^2)$.

1.2.3 Symmetry

- $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.
- $f(n) = \Omega(g(n))$ implies $g(n) = \Omega(f(n))$.
- $f(n) = \Theta(g(n))$ implies $g(n) = \Theta(f(n))$.
- $f(n) = o(g(n))$ implies $g(n) = \omega(f(n))$.
- $f(n) = \omega(g(n))$ implies $g(n) = o(f(n))$.

For example, $n^2 = O(n^3)$ implies $n^3 = \Omega(n^2)$.

1.2.4 Transpose Symmetry

- $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
- $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
- $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.
- $f(n) = \omega(g(n))$ if and only if $g(n) = o(f(n))$.

For example, $n^2 = O(n^3)$ if and only if $n^3 = \Omega(n^2)$.

1.2.5 sum and maximum

$$f_1(n) + f_2(n) + \cdots + f_k(n) = \Theta(\max(f_1(n), f_2(n), \ldots, f_k(n)))$$

where k is a constant positive integer.
Let $f_j(n) = j$, $k = n$, then

$$f_1(n) + f_2(n) + \cdots + f_k(n) = n(n+1)/2 = \Theta(n^2)$$

1.2.6 Running time hierarchy

- logarithmic: $O(\log n)$
- linear: $O(n)$
- $n \log n$: $O(n \log n)$
- quadratic: $O(n^2)$
- polynomial: $O(n^k)$
- exponential: $O(c^n)$
- constant: $O(1)$
- superconstant: $\omega(1)$
- sublinear: $o(n)$
- superlinear: $\omega(n)$
- superpolynomial: $\omega(n^k)$
- subexponential: $o(c^n)$

1.3 Expect of algorithms

Correctness: An algorithm is correct if it halts with the correct output for every input instance.
Termination: An algorithm is terminating if it halts for every input instance.
Efficiency: An algorithm is efficient if it halts with the correct output for every input instance and runs in polynomial time.

Chapter 2

Recursion and Divide and Conquer techniques

2.1 Finding Majority in array

The pesudocode of the algorithm is shown in Algorithm 2.1.

Algorithm 1 Finding Majority in array

```
procedure MAJORITY(A)
    n ← length of A
    if n = 0 then
        return −1
    end if
    if n = 1 then
        return A[1]
    end if
    if n1 and n is odd then

    end if
    Array B of size n/2
    set j=0
    for i = 1 to n/2 do
        if A[2i − 1] = A[2i] then
            B[j] ← A[2i − 1]
            j ← j + 1
        end if
    end for
    m ← MAJORITY(B)
    count ← 0
    for i = 1 to n do
        if A[i] = m then
            count ← count + 1
        end if
    end for
    if count > n/2 then
        return m
    else
        return −1
    end if
end procedure
```

Correctness:
Lemma: If A has a majority element, then the majority element of A is also the majority element of B.
Base case: $n = 1$, the majority element is $A[1]$.
Induction hypothesis: Assume that the lemma is true for $n = k$, we will prove that the lemma is true for $n = k + 1$.
Induction step: If A has a majority element, then the majority element of A is also the majority element of B.
Case 1 (A has a majority element m): Then by the lemma, it is also the majority element of B. Then m appears more than $k/2$ times in B. Then m appears more than $(k + 1)/2$ times in A.
Case 2 (A has no majority element): Then B has no majority element. Then A has no majority element.
Proof the lemma:
proof by contradiction. Assume that A has a majority element m and B has a majority element m', but $m \neq m'$.
Let x be the numbers of occurrence of m in A.
Let y be the numbers of occurrence of m' in B.
Then $2y$ times from pairs that are represented in B by a value different from m', and $x - 2y$ times, since each occurrence of m in A that is not paired with another occurrence of m in A is paired with an occurrence of m' in B.
In total, this gives $2y + x - 2y = x$ occurrences of m in A, which is a contradiction.
Running time:
Recursive formula for the running time:

$$T(n) \leq T(n/2) + cn$$

where c is a constant.
The solution to the recurrence is $T(n) = O(n)$.

2.2 Searching in logarithmic time

Searching faster with BinarySearch.
It is a particular case of the divide-and-conquer paradigm.
Input: A sorted array A of n elements and a value x.
Output: An index i such that $A[i] = x$ or the special value -1 if x does not appear in A.
Pseudocode is shown in Algorithm 2.2.

Algorithm 2 BinarySearch

```
procedure BinarySearch(x, i, j)
    if i = j then
        if A[i] = x then
            return i
        else
            return −1
        end if
    else
        if x = A[⌊(i + j)/2⌋] then
            return ⌊(i + j)/2⌋
        else if x < A[⌊(i + j)/2⌋] then
            return BinarySearch(x, i, ⌊(i + j)/2⌋)
        else
            return BinarySearch(x, ⌊(i + j)/2⌋ + 1, j)
        end if
    end if
end procedure
```

Running time:
The number of comparisons performed by BinarySearch is:

$$T(n) \leq T(n/2) + 4$$

Keep calculate:

$$\begin{aligned}
T(n) &\leq T(n/2) + 4 \\
&\leq T(n/4) + 4 + 4 \\
&\leq T(n/8) + 4 + 4 + 4 \\
&\leq T(n/2^k) + 4k \\
&\leq T(n/2^{\log(n-1)}) + 4\log(n-1) \\
&= T(2) + 4(\log n - 1) \\
&\leq 4\log n - 4 \\
&= 4\log n
\end{aligned}$$

proof $T(n) \leq 4\log n$:
Base case: $n = 1, T(1) = 0 \leq 4\log 1 = 0$.
Induction hypothesis: Assume that the lemma is true for $n = k$, we will prove that the lemma is true for $n = k + 1$.
Induction step: $T(k+1) \leq 4\log(k+1)$.

$$\begin{aligned}
T(k+1) &\leq T(k/2) + 4 \\
&\leq 4\log(k/2) + 4 \\
&= 4\log k - 4 + 4 \\
&= 4\log k \\
&\leq 4\log(k+1)
\end{aligned}$$

Memory usage:
The memory usage of BinarySearch is:

$$M(n) = O(\log n)$$

Comparing BinarySearch and LinearSearch:

$$T_{\text{BinarySearch}}(n) = O(\log n)$$
$$T_{\text{LinearSearch}}(n) = O(n)$$
$$T_{\text{BinarySearch}}(n) = O(\log n) < O(n) = T_{\text{LinearSearch}}(n)$$
$$M_{\text{BinarySearch}}(n) = O(\log n) < O(1) = M_{\text{LinearSearch}}(n)$$

2.3 Running time of Divide and Conquer algorithms

The Master Theorem:
Suppose that $T(n)$ satisfies the recurrence:

$$T(n) \leq aT(n/b) + cn^d$$

where $a \geq 1$, $b > 1$, $c > 0$ and $d \geq 0$ are constants.
Then $T(n)$ has the following asymptotic bounds:

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

This theorem is useful for solving recurrences of the form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$, $b > 1$ and $f(n)$ is an asymptotically positive function.
Example:

$$T(n) = 8T(n/2) + 100n^2$$

$a = 8$, $b = 2$, $f(n) = 100n^2$, $d = 2$, $\log_b a = \log_2 8 = 3$.
$d = 2 < \log_b a = 3$, so $T(n) = O(n^{\log_b a}) = O(n^3)$.

2.4 Finding piar of points closest to each other

Input: A set P of n points in the plane.
Output: The pair of points in P that are closest to each other.
Pseudocode is shown in Algorithm 2.4. **Running time**:

Algorithm 3 ClosestPair

procedure CLOSESTPAIR($P_1, \dots, P_n$)
  Construct P_x and P_y. P_x is sorted by x-coordinate, P_y is sorted by y-coordinate.
  return CLOSESTPAIRREC(P_x, P_y)
end procedure

Algorithm 4 ClosestPairRec

procedure CLOSESTPAIRREC(P_x, P_y)
  if $|P_x| = |P_y| \leq 3$ **then**
    For each pair of points (P_i, P_j), compute $d(P_i, P_j)$
    return the pair of points with the smallest distance
  end if
  Construct Q_x, Q_y, R_x and R_y.
  (l_1, l_2) = CLOSESTPAIRREC(Q_x, Q_y)
  (r_1, r_2) = CLOSESTPAIRREC(R_x, R_y)
  $\delta = \min\{d(l_1, l_2), d(r_1, r_2)\}$
  x^* = the largest x-coordinate in Q_x
  $L = (x, y) : x = x^*$
  $S = \{p \in P : p \in L$ and p is within δ of $L\}$
  Construct S_v
  for $p \in S$ **do**
    Let q be the point in S_v closest to p
    if $d(p, q) < \delta$ **then**
      $\delta = d(p, q)$
      (s_1, s_2) = (p, q)
    end if
  end for
  if $d(s_1, s_2) < \min\{d(l_1, l_2), d(r_1, r_2)\}$ **then**
    return (s_1, s_2)
  end if
  if $d(l_1, l_2) < d(r_1, r_2)$ **then**
    return (l_1, l_2)
  else
    return (r_1, r_2)
  end if
end procedure

$T(n) \leq 2T(n/2) + O(n \log n) = O(n \log n)$
Example:

Chapter 3

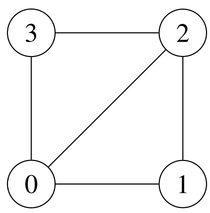
Graph Algorithms

3.1 Graph Definitions

Graph: A graph G consists of a set V of vertices and a set E of edges, where each edge is associated with a pair of vertices.
Directed Graph: A directed graph G consists of a set V of vertices and a set E of directed edges, where each directed edge is associated with an ordered pair of vertices.
Undirected Graph: An undirected graph G consists of a set V of vertices and a set E of undirected edges, where each undirected edge is associated with an unordered pair of vertices.
Neighbours of a vertex v: Set of vertices that are connected to v by an edge.
Degree of a vertex v: number of neighbours of v, denoted by $deg(v)$.
Path: A sequence of (non-repeating) nodes with consecutive nodes being connected by an edge.
length = node count - 1 = edge count.
Distance between two nodes: The number of edges in the shortest path between the two nodes.
Graph diameter: The maximum distance between any two nodes in the graph.

Lines,cycles,trees and cliques:
Line: A graph with n vertices and $n-1$ edges.
Cycle: A graph with n vertices and n edges.
cliques: A graph with n vertices and $n(n-1)/2$ edges.
Tree: A graph with n vertices and $n-1$ edges.

Graph representations:
Adjacency matrix: A $n \times n$ matrix A where $A_{ij} = 1$ if there is an edge between i and j, and $A_{ij} = 0$ otherwise.
examples of adjacency matrices:
Given the following graph:

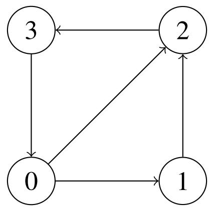


The adjacency matrix is:

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Adjacency matrix for directed graphs: A $n \times n$ matrix A where $A_{ij} = 1$ if there is an edge from i to j, and $A_{ij} = 0$ otherwise.

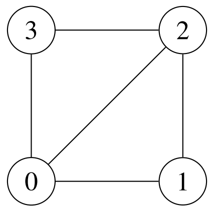
examples of adjacency matrices for directed graphs:
Given the following graph:



The adjacency matrix is:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

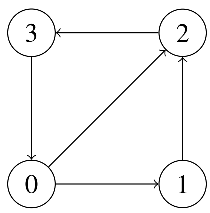
Adjacency list: A list of lists, where the ith list contains the neighbours of vertex i.
Given the following graph:



The adjacency list is:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 2 & \\ 0 & 1 & 3 \\ 0 & 2 & \end{bmatrix}$$

Adjacency list for directed graphs: A list of lists, where the ith list contains the neighbours of vertex i.
Given the following graph:



The adjacency list is:

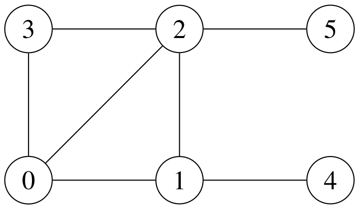
$$\begin{bmatrix} 1 & 2 \\ 2 & \\ 3 & \\ 0 & \end{bmatrix}$$

Adjacency matrix vs adjacency list:

Adjacency matrix	Adjacency list
$O(1)$ to check if there is an edge between i and j	$O(min(deg(i), deg(j)))$ to check if there is an edge between i and j
$O(n)$ to find the neighbours of i	$O(deg(j))$ to find the neighbours of i
$O(n^2)$ space	$O(n + m)$ space

3.2 Depth-first search

Depth-first search: A graph search algorithm that explores the neighbours of a vertex before exploring the neighbours of its neighbours.
example of depth-first search:



The depth-first search sequence is:

$$0, 1, 2, 3, 5, 4$$

Depth-first search algorithm:

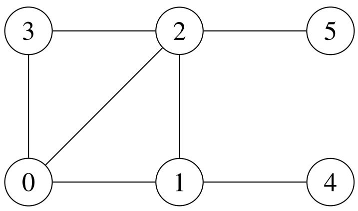
Algorithm 5 Depth-first search algorithm

procedure DFS(G, v)
    for $e \in V$ **do**
        if e is unexplored **then**
            u = head of e
            if u is unexplored **then**
                e is a tree edge
                DFS(G, u)
            else
                e is a back edge
            end if
        end if
    end for
end procedure

Running time of depth-first search: $O(n + m)$

3.3 Breadth-first search

Breadth-first search: A graph search algorithm that explores the neighbours of a vertex before exploring the neighbours of its neighbours.
exaqmple of breadth-first search:



The breadth-first search sequence starting from vertex 0 is 0, 1, 2, 3, 4, 5.
Breadth-first search algorithm:

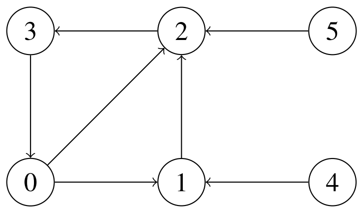
Algorithm 6 Breadth-first search algorithm

procedure BFS(G, s)
    initial empty list L
    $L \leftarrow s$
    $i \leftarrow 0$
    while $L[i] \neq \emptyset$ **do**
        $L_{i+1} \leftarrow emptylist$
        for $v \in L[i]$ **do**
            for edges (e) incident to v **do**
                if e is unexplored **then**
                    $w \leftarrow$ the other end of e
                    if w is unexplored **then**
                        label e as a tree edge
                        add w to L_{i+1}
                    else
                        label e as a cross edge
                    end if
                end if
            end for
        end for
        $i \leftarrow i + 1$
    end while
end procedure

Running time of breadth-first search: $O(n + m)$

3.4 Strong Connectivity

Directed graph: A graph where the edges have a direction.
Examples:



DFS and BFS on directed graphs:
Very similar to undirected graphs, except that we only consider edges that go out of a vertex.
Running time is $O(n + m)$
For example graph above the DFS sequence is 0, 1, 2, 3.
The BFS sequence is 0, 1, 2, 3.

3.4.1 Connectivity

Weak connectivity:If we ignore the direction for all edges, there would be a pah from any vertex to any other vertex.
Strong Connectivity:For every two nodes u and v, there is a path from u to v and a path from v to u.

3.4.2 Mutual Reachability

Two nodes u and v are mutually reachable if there is a path from u to v and a path from v to u.
Strong connectivity:For every pair of nodes u and v, these two nodes are mutually reachable.
Transitivity:If u is mutually reachable with v and v is mutually reachable with w, then u is mutually reachable with w.

3.4.3 Testing strong connectivity

Algorithm 7 Testing strong connectivity

```
procedure TESTSTRONGCONNECTIVITY(G)
    define G^R to be the graph with the same vertices as G but with all edges reversed
    Select a node s in G
    BFS(G, s),BFS(G^R, s)
    for each node v do
        if v is unexplored in either BFS then
            return False
        end if
    end for
    return True
end procedure
```

3.5 Testing bipartiteness

Bipartite graph: A graph $G = (V, E)$ is bipartite if any only if the vertices can be partitioned into two sets V_1 and V_2 such that every edge has one end in V_1 and the other end in V_2.
A Graph $G = (V, E)$ is bipartite if and only if it has no odd cycles.(odd cycle: a cycle with odd number of edges)
Testing bipartiteness:
Given a graph $G = (V, E)$, we want to test if G is bipartite.
Given a graph $G = (V, E)$, decide if it is 2-colourable.
Given a graph $G = (V, E)$, decide if it has an odd cycle.
Colouring the nodes It is quite familiar with BFS:

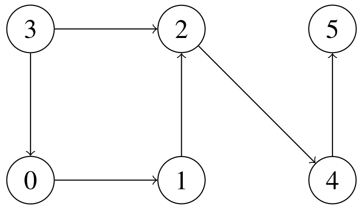
Algorithm 8 Colouring the nodes

```
procedure COLOURING(G, s)
    initial empty list L
    initial empty list C
    L ← s
    C[s] ← red
    i ← 0
    while L[i] ≠ ∅ do
        L_{i+1} ← emptylist
        for v ∈ L[i] do
            for edges (e) incident to v do
                if e is unexplored then
                    w ← the other end of e
                    if w is unexplored then
                        label e as a tree edge
                        add w to L_{i+1}
                        if i + 1 is odd then
                            C[w] ← green
                        else
                            C[w] ← red
                        end if
                    else
                        label e as a cross edge
                        if C[v] = C[w] then
                            return False
                        end if
                    end if
                end if
            end for
        end for
        i ← i + 1
    end while
    for e(v, w) ∈ G do
        if C[v] = C[w] then
            return False
        end if
    end for
    return True
end procedure
```

Running time of colouring the nodes: $O(n + m)$
Correctness of colouring the nodes:
Proof by contradiction.
Suppose that G is not bipartite.
Then G has an odd cycle.
Suppose to the contrary that the algorithm return True.
That means that the algorithm did not detect the odd cycle.

3.6 DAGs and Topological Ordering

DAG: A directed acyclic graph (DAG) is a directed graph with no directed cycles.
examples of DAGs:



Topological ordering:Given a graph$G = (V, E)$, a topological ordering of G is an ordering of the nodes $u_1, u_2, \ldots, u_n$ such that for every edge (u_i, u_j), we have $i < j$.
Intutively, a topological ordering is an ordering of the nodes such that every edge goes from left to right.
example of topological ordering based on given graph above:

$$3, 0, 1, 2, 4, 5$$

Topological ordering implies DAG:

- If G has a topological ordering, then G is a DAG.
- Suppose by contradiction that G has a topological ordering $u_1, u_2, \ldots, u_n$ but G also has a cycle C.
- Let u_j be the smallest element of C in the topological ordering.
- Let u_i be its predecessor in C.
- u_i must appear before u_j in the topological ordering.
- This contradicts the fact that u_j is the smallest element of C in the topological ordering.

DAG implies topological ordering:
Proof by induction: Base case: If G has one or two nodes, then G has a topological ordering.
Induction steps: Assume that a DAG up to