CS 170, Fall 2022

Contents

1. Big-O Notation

Definition 1.1

Let $f(n)$ and $g(n)$ be functions from positive integers to positive reals. We say $f = O(g)$ if there is a constant $c > 0$ such that $f(n) \leq cg(n)$

Saying $f = O(g)$ is a very loose analog of "$f \leq g$."

Definition 1.2

$$f = \Omega(g) \iff g = O(f)$$
$$f = \Theta(g) \iff f = O(g) \ \wedge \ f = \Omega(g)$$

Saying $f = \Omega(g)$ is a very loose analog of "$f \geq g$," and therefore $f = \Theta(g)$ means that f and g takes, in average, the time to run as the input size grows (g encloses f both from above and below).

Example 1.3

TODO

2. Divide-and-conquer Algorithms

2.a. Multiplication

Definition 2.1 (Integer Multiplication)

A divide-and-conquer algorithm for integer multiplication is defined as follows:

```
function mul(x(0b[1...k]), y(0b[1...h]))
  %Input: Positive integers x, y in binary
  %Output: x times y

  n = max(size of x, size of y)
  if n == 1: return x × y

  x_L, x_R = x(0b[1...⌈n/2⌉]), x(0b[⌊n/2⌋...n])
  y_L, y_R = y(0b[1...⌈n/2⌉]), y(0b[⌊n/2⌋...n])

  P_1 = mul(x_L, y_L)
  P_2 = mul(x_R, y_R)
  P_3 = mul(x_L + x_R, y_L + y_R)
  return P_1 × 2^n + (P_3 − P_1 − P_2) × 2^{n/2} + P_2
```

Where $0b[1...k]$ denotes the binary string representing a number.

Each call of *mul* has three recursive calls, inputs of which are half the size of the original inputs, and the base cases (x times y) take constant time. Therefore we conclude that the time taken by this algorithm is

$$T(n) = 3T(n/2) + O(n)$$

Apply the Master Algorithm in Chap 2.b, we conclude that the time complexity of this algorithm is

$$T(n) \in \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$$

2.b. Recurrence Relations

Theorem 2.2 (Master Algorithm)

If $T(n) = aT(n/b) + cn^k$ and $T(1) = c$ for some constants a, b, c and k, then

$$T(n) \in \begin{cases} \Theta(n^k) & if\ a < b^k \\ \Theta(n^k \log n) & if\ a = b^k \\ \Theta(n^{\log_b a}) & if\ a > b^k \end{cases}$$

2.c. Mergesort

Definition 2.3 (Mergesort)

The Mergesort algorithm is defined as follows:

```
function mergesort(a[1...n])
  %Input: An array of numbers a[1...n]
  %Output: Sorted array a

  if n>1:
    return merge(mergesort(a[1...⌊n/2⌋]), mergesort(a[⌊n/2⌋+1...n]))
  else:
    return a

function merge(x[1...k], y[1...h])
  %Input: Two arrays of numbers (x[1...k], y[1...h])
  %Output: An array of numbers in x and y in ascending order

  if k=0: return y
  if l=0: return x
  if x[1] <= y[1]:
    return x[1] ∘ merge(x[2...k], y[1...h])
  else:
    return y[1] ∘ merge(x[1...k], y[2...h])
```

Where $\circ$ denotes concatenation.

The *merge* function above does a constant amount of work (concatenating two arrays) per recursive call, for a total running time of $O(k+h)$. Thus the calls to *merge* in *mergesort* are linear, we conclude that the overall time taken by *mergesort* is

$$T(n) = 2T(n/2) + O(n)$$

Recall the Master Algorithm in Chap 2.b, we conclude that the time complexity of this algorithm is

$$T(n) \in \Theta(n \log n)$$

Remark 2.4

$n \log n$ is the lower bound for sorting, and therefore *mergesort* is optimal.

Proof. Sorting algorithms can be depicted as trees that each non-leaf node represents a comparison between two elements, and each leaf denotes a permutation of the input array (and thus a binary search tree since each non-leaf nodes have two children). Consider such tree that sorts an array $a[1...n]$. The total number of the leaves is $n!$. A binary tree of depth d has at most 2^d leaves. Therefore, the depth of the tree and the complexity of this algorithm should be at least $\log(n!)$, which is the worst case of this algorithm. Since $\log(n!) \leq cn \log(n)$, we conclude that $n \log(n)$ is optimal for sorting algorithms. ∎

2.d. Medians

Definition 2.5 (selection)
A randomized divide-and-conquer algorithm for selection is defined as follows

2.e. Matrix Multiplication

Definition 2.6

2.f. Fast Fourier Transform

3. Decomposition of Graphs

3.a. DFS in Undirected Graphs

Definition 3.1 (explore)

Finding all nodes reachable from a particular node.

```
procedure explore(G, v):
  % Input: Graph G = (V, E), v a node in V
  % Output: u.visited is set true for all node u reachable from v
  v.visited = true
  previsit(v)
  foreach (v, u) in E:
    if not u.visited: explore(G, u)
  postvisit(v)
```

Where previsit(v) and postvisit(v) denotes the "time" τ before and after v is explored, respectively.

Definition 3.2 (DFS)

Based on Definition of explore, a DFS procedure is as follows:

```
procedure DFS(G)
  % Input: Graph G = (V, E)
  forall v in V:
    if not v.visited: explore(v)
```

Definition 3.3 (ordering)

The previsit and postvisit ordering is defined as follows:

```
procedure previsit(v)
  pre[v] = clock
  clock += 1
procedure postvisit(v)
  post[v] = clock
  clock += 1
```

Remark 3.4

The implementation of a DFS uses a stack and DFS's runtime is $O(|V| + |E|)$.

3.b. DFS in Directed Graphs

Definition 3.5 (Type of Edges)

There four types of edges:

- Tree edges are part of the DFS forest
- Forward edges lead to a nonchild descendant
- Back edges lead to a not-direct ancestor
- Cross edges lead to a node that is neither descendant nor ancestor, a node that has already been completely explored.

An edge (u, v) in E is:

- Forward if pre(u) < pre(v) < post(v) < post(u)
- Back if pre(v) < pre(u) < post(u) < post(v)
- Cross if pre(v) < post(v) < pre(u) < post(u)

Definition 3.6

A directed graph has a cycle iff its DFS reveals a back edge. If the DFS of a directed graph reveals no back edge, the graph is a Directed Acyclic Graph (DAG).

Theorem 3.7

All DAG can be linearized (topologically sorted). That is, if G(V, E) is a DAG with (u, v) in E, then post(u) > post(v).

Theorem 3.8

All DAG must have at least one source (a node with no ingoing edges) and at least one sink (a node with no outgoing edges).

Theorem 3.9

A DAG, after delete one of its sources, is still a DAG.

Theorem 3.10

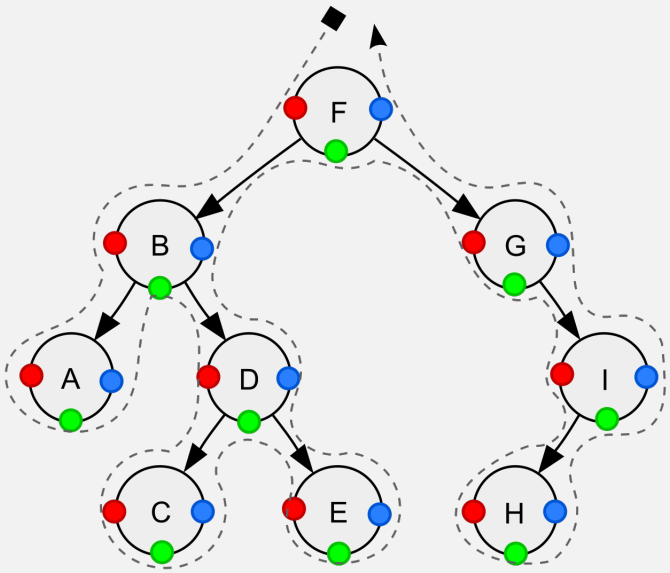
A DAG have one or more possible linearizations. The following algorithm,

```
procedure linearize(G)
  while G:
    find a source s in G
    pop s
```

Is guaranteed to give a linearization.

Remark 3.11

In a DF traverse of a binary tree, the visiting order of nodes can be found by labeling the graph like follows:



Where red dots denote preorder, green dots denote in-order, and blue dots denote post-order.

3.c. Strongly Connected Components

Definition 3.12 (Node connectivity)

Two nodes u and v of a directed graph are connected if there is a path from u to v and a path from v to u.

Definition 3.13 (Strongly Connected)

Disjoint sets in V partitioned by the connectivity relations are strongly connected components.

Theorem 3.14

All directed graph is a DAG of its strongly connected components.

4. Paths in Graphs

4.a. Distances

4.b. BFS

4.c. Lengths on Edges

4.d. Dijkstra's

4.e. Priority Queue Implementation

4.f. Shortest Paths in the Presence of Negative Edges

4.g. Shortest Paths in DAGs