ECE 374 B Algorithms: Cheatsheet

1 Recursion

Simple recursion

Definitions

- **Reduction:** solve one problem using the solution to another.
- **Recursion:** a special case of reduction - reduce problem to a *smaller* instance of *itself* (self-reduction).
  - Problem instance of size n is reduced to *one or more* instances of size $n-1$ or less.
  - For termination, problem instances of small size are solved by some other method as *base cases*

Tower of Hanoi

Arguably the most famous example of recursion. The goal is to move n disks one at a time from the first peg to the last peg.

Pseudocode: Tower of Hanoi

```
Hanoi (n, src, dest, tmp):
  if (n > 0) then
    Hanoi (n − 1, src, tmp, dest)
    Move disk n from src to dest
    Hanoi (n − 1, tmp, dest, src)
```

Recurrences

Suppose you have a recurrence of the form $T(n) = rT(n/c) + f(n)$.

The *master theorem* gives a good asymptotic estimate of the recurrence. If the work at each level is:

Decreasing:	$rf(n/c) = \kappa f(n)$ where $\kappa < 1$	$T(n) = O(f(n))$
Equal:	$rf(n/c) = f(n)$	$T(n) = O(f(n) \cdot \log_c n)$
Increasing:	$rf(n/c) = Kf(n)$ where $K > 1$	$T(n) = O(n^{\log_c r})$

Some useful identities:

- Sum of integers: $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$
- Geometric series closed-form formula: $\sum_{k=0}^{n} ar^k = a\frac{1-r^{n+1}}{1-r}$
- Logarithmic identities: $\log(ab) = \log a + \log b$, $\log(a/b) = \log a - \log b$, $a^{\log_c b} = b^{\log_c a}$ $(a, b, c > 1)$, $\log_a b = \log_c b / \log_c a$.

Backtracking

Backtracking is the algorithm paradigm involving guessing the solution to a single step in some multi-step process and recursing backwards if it doesn't lead to a solution. For instance, consider the longest increasing subsequence (LIS) problem. You can either check all possible subsequences:

Pseudocode: LIS - Naive enumeration

```
algLISNaive(A[1..n]):
  maxmax = 0
  for each subsequence B of A do
    if B is increasing and |B| > max then
      max = |B|
  return max
```

On the other hand, we don't need to generate every subsequence; we only need to generate the subsequences that are increasing:

Pseudocode: LIS - Backtracking

```
LIS_smaller(A[1..n], x):
  if n = 0 then return 0
  max = LIS_smaller(A[1..n − 1], x)
  if A[n] < x then
    max = max {max, 1 + LIS_smaller(A[1..(n − 1)], A[n])}
  return max
```

Divide and conquer

Divide and conquer is an algorithm paradigm involving the decomposition of a problem into the same subproblem, solving them separately and combining their results to get a solution for the original problem.

Sorting algorithms

Algorithm	Runtime	Space
Mergesort	$O(n \log n)$	$O(n \log n)$ $O(n)$ (if optimized)
Quicksort	$O(n^2)$ $O(n \log n)$ if using MoM	$O(n)$

Karatsuba's algorithm

We can divide and conquer multiplication like so:

$$bc = 10^n b_L c_L + 10^{n/2}(b_L c_R + b_R c_L) + b_R c_R.$$

We can rewrite the equation as:

$$\begin{aligned} bc = b(x)c(x) = (b_L x + b_R)(c_L x + c_R) &= (b_L c_L)x^2 \\ &+ ((b_L + b_R)(c_L + c_R) - b_L c_L - b_R c_R)\, x \\ &+ b_R c_R, \end{aligned}$$

Its running time is $O(n^{\log_2 3}) = O(n^{1.585})$.

Linear time selection

The *median of medians* (MoM) algorithms give a element that is larger than $\frac{3}{10}$'s and smaller than $\frac{7}{10}$'s of the array elements. This is used in the linear time selection algorithm to find element of rank k.

Pseudocode: Quickselect with median of medians

```
Median-of-medians (A, i):
  sublists = [A[j:j+5] for j ← 0, 5, . . . , len(A)]
  medians = [sorted (sublist)[len (sublist)/2]
            for sublist ∈ sublists]

  // Base case
  if len (A) ≤ 5 return sorted (a)[i]

  // Find median of medians
  if len (medians) ≤ 5
    pivot = sorted (medians)[len (medians)/2]
  else
    pivot = Median-of-medians (medians, len/2)

  // Partitioning step
  low = [j for j ∈ A if j < pivot]
  high = [j for j ∈ A if j > pivot]

  k = len (low)
  if i < k
    return Median-of-medians (low, i)
  else if i > k
    return Median-of-medians (low, i-k-1)
  else
  return pivot
```

Dynamic programming

Dynamic programming (DP) is the algorithm paradigm involving the computation of a recursive backtracking algorithm iteratively to avoid the recomputation of any particular subproblem.

Longest increasing subsequence

The longest increasing subsequence problem asks for the length of a longest increasing subsequence in a unordered sequence, where the sequence is assumed to be given as an array. The recurrence can be written as:

$$LIS(i,j) = \begin{cases} 0 & \text{if } i = 0 \\ LIS(i-1,j) & \text{if } A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1,j) \\ 1 + LIS(i-1,i) \end{cases} & \text{else} \end{cases}$$

Pseudocode: LIS - DP

LIS-Iterative$(A[1..n])$:
 $A[n+1] = \infty$
 for $j \leftarrow 0$ to n
  if A[i] $\leq$ A[j] **then** $LIS[0][j] = 1$

 for $i \leftarrow 1$ to $n-1$ **do**
  for $j \leftarrow i$ to $n-1$ **do**
   if $A[i] \geq A[j]$
    $LIS[i,j] = LIS[i-1,j]$
   else
    $LIS[i,j] = \max\{LIS[i-1,j],$
     $1 + LIS[i-1,i]\}$
 return $LIS[n, n+1]$

Edit distance

The edit distance problem asks how many edits we need to make to a sequence for it to become another one. The recurrence is given as:

$$\text{Opt}(i,j) = \min \begin{cases} \alpha_{x_i y_j} + \text{Opt}(i-1,j-1), \\ \delta + \text{Opt}(i-1,j), \\ \delta + \text{Opt}(i,j-1) \end{cases}$$

Base cases: $\text{Opt}(i,0) = \delta \cdot i$ and $\text{Opt}(0,j) = \delta \cdot j$

Pseudocode: Edit distance - DP

$EDIST(A[1..m], B[1..n])$
 for $i \leftarrow 1$ to m **do** $M[i,0] = i\delta$
 for $j \leftarrow 1$ to n **do** $M[0,j] = j\delta$

 for $i = 1$ to m **do**
  for $j = 1$ to n **do**

$$M[i][j] = \min \begin{cases} COST[A[i]][B[j]] \\ \qquad + M[i-1][j-1], \\ \delta + M[i-1][j], \\ \delta + M[i][j-1] \end{cases}$$

2 Graph algorithms

Graph basics

A graph is defined by a tuple $G = (V,E)$ and we typically define $n = |V|$ and $m = |E|$. We define (u,v) as the edge from u to v. Graphs can be represented as **adjacency lists**, or **adjacency matrices** though the former is more commonly used.

- *path*: sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $v_i v_{i+1} \in E$ for $1 \leq i \leq k-1$. The length of the path is $k-1$ (the number of edges in the path). *Note:* a single vertex u is a path of length 0.
- *cycle*: sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$ and $(v_k, v_1) \in E$. A single vertex is not a cycle according to this definition.
 Caveat: Sometimes people use the term cycle to also allow vertices to be repeated; we will use the term *tour*.
- A vertex u is *connected* to v if there is a path from u to v.
- The *connected component* of u, con(u), is the set of all vertices connected to u.
- A vertex u can *reach* v if there is a path from u to v. Alternatively v can be reached from u. Let **rch**(u) be the set of all vertices reachable from u.

Directed acyclic graphs

Directed acyclic graphs (dags) have an intrinsic ordering of the vertices that enables dynamic programming algorithms to be used on them.
A *topological ordering* of a dag $G = (V,E)$ is an ordering $\prec$ on V such that if $(u,v) \in E$ then $u \prec v$.

Pseudocode: Kahn's algorithm

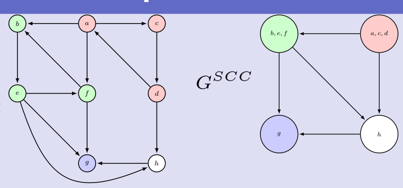
Kahn$(G(V,E), u)$:
toposort$\leftarrow$empty list
for $v \in V$:
 in$(v) \leftarrow |\{u \mid u \to v \in E\}|$
while $v \in V$ that has in$(v) = 0$:
 Add v to end of toposort
 Remove v from V
 for v in $u \to v \in E$:
  in$(v) \leftarrow$ in$(v) - 1$
return toposort

Running time: $O(n+m)$

- A dag may have multiple topological sorts.
- A topological sort can be computed by DFS, in particular by listing the vertices in decreasing post-visit order.

Strongly connected components

- Given G, u is *strongly connected* to v if $v \in$ rch(u) *and* $u \in$ rch(v).
- A *maximal* group of G: vertices that are all strongly connected to one another is called a strong component.



Pseudocode: Metagraph - linear time

Metagraph$(G(V,E))$:
 Compute rev(G) by brute force
 ordering $\leftarrow$ reverse postordering of V in rev(G)
  by **DFS**(rev$(G), s$) for any vertex s
 Mark all nodes as unvisited
 for each u in ordering **do**
  if u is not visited and $u \in V$ **then**
   $S_u \leftarrow$ nodes reachable by u by **DFS**(G,u)
   Output S_u as a strong connected component
   $G(V,E) \leftarrow G - S_u$

Running time: $O(m+n)$

DFS and BFS

Pseudocode: Explore (DFS/BFS)

```
Explore(G,u):
  for i ← 1 to n:
    Visited[i] ← False
  Add u to ToExplore and to S
  Visited[u] ← True
  Make tree T with root as u
  while ToExplore is non-empty do
    Remove node x from ToExplore
    for each edge (x, y) in Adj(x) do
      if Visited[y] = False
        Visited[y] ← True
        Add y to ToExplore, S, T (with x as parent)
```

- If B is a queue, *Explore* becomes BFS.
- If B is a stack, *Explore* becomes DFS.

Running time: $O(m + n)$

Pre/post numbering

Pre and post numbering aids in analyzing the graph structure. By looking at the numbering we can tell if a edge (u, v) is a:

- *Forward edge*: pre$(u) <$ pre$(v) <$ post$(v) <$ post(u)
- *Backward edge*: pre$(v) <$ pre$(u) <$ post$(u) <$ post(v)
- *Cross edge*: pre$(u) <$ post$(u) <$ pre$(v) <$ post(v)

Minimum Spanning Tress

- Tree = undirected graph in which any two vertices are connected by exactly one path.
- Sub-graph H of G is *spanning* for G, if G and H have same connected components.
- A minimum spanning tree is composed of all the safe edges in the graph
- An edge $e = (u, v)$ is a *safe* edge if there is some partition of V into S and $V \setminus S$ and e is the unique minimum cost edge crossing S (one end in S and the other in $V \setminus S$).
- An edge $e = (u, v)$ is an *unsafe* edge if there is some cycle C such that e is the unique maximum cost edge in C.

Pseudocode: Boruvka's algorithm: $O(m\log(n))$

```
T is ∅ (* T will store edges of a MST *)
while T is not spanning do
  X ← ∅
  for each connected component S of T do
    add to X the cheapest edge between S and V \ S
  Add edges in X to T
return the set T
```

Running time: $O(m\log(n))$

Pseudocode: Kruskal's algorithm: $(m+n)\log(m)$ (using Union-Find structure)

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty do
  pick e = (u, v) ∈ E of minimum cost
  if u and v belong to different sets
    add e to T
    merge the sets containing u and v
return the set T
```

Running time: $O((m + n)\log(m))$ if using union-find data structure

Pseudocode: Prim's algorithm: $(n)\log(n) + m$ (using Priority Queue)

```
T ← ∅, S ← ∅, s ← 1
∀v ∈ V(G) : d(v) ← ∞, p(v) ← ∅
d(s) ← 0
while S ≠ V do
  v = arg min_{u∈V\S} d(u)
  T = T ∪ {vp(v)}
  S = S ∪ {v}
  for each u in Adj(v) do
    d(u) ← min { d(u), c(vu) }
    if d(u) = c(vu) then
      p(u) ← v
return T
```

Running time: $O(n\log(n) + m)$ if using Fibonacci heaps

Shortest paths

Dijkstra's algorithm:
Find minimum distance from vertex s to **all** other vertices in graphs *without* negative weight edges.

Pseudocode: Dijkstra

```
for v ∈ V do
  d(v) ← ∞
X ← ∅
d(s, s) ← 0
for i ← 1 to n do
  v ← arg min_{u∈V−X} d(u)
  X = X ∪ {v}
  for u in Adj(v) do
    d(u) ← min {(d(u), d(v) + ℓ(v, u))}
return d
```

Running time: $O(m+n\log n)$ (if using a Fibonacci heap as the priority queue)

Bellman-Ford algorithm:
Find minimum distance from vertex s to **all** other vertices in graphs *without* negative cycles. It is a DP algorithm with the following recurrence:

$$d(v,k) = \begin{cases} 0 & \text{if } v = s \text{ and } k = 0 \\ \infty & \text{if } v \neq s \text{ and } k = 0 \\ \min \begin{cases} \min_{uv \in E}\{d(u,k-1) + \ell(u,v)\} \\ d(v,k-1) \end{cases} & \text{else} \end{cases}$$

Base cases: $d(s, 0) = 0$ and $d(v, 0) = \infty$ for all $v \neq s$.

Pseudocode: Bellman-Ford

```
for each v ∈ V do
  d(v) ← ∞
d(s) ← 0

for k ← 1 to n − 1 do
  for each v ∈ V do
    for each edge (u, v) ∈ in(v) do
      d(v) ← min{d(v), d(u) + ℓ(u, v)}

return d
```

Running time: $O(nm)$

Floyd-Warshall algorithm:
Find minimum distance from *every* vertex to *every* vertex in a graph *without* negative cycles. It is a DP algorithm with the following recurrence:

$$d(i,j,k) = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } (i,j) \notin E \text{ and } k = 0 \\ \min \begin{cases} d(i,j,k-1) \\ d(i,k,k-1) + d(k,j,k-1) \end{cases} & \text{else} \end{cases}$$

Then $d(i, j, n - 1)$ will give the shortest-path distance *from* i *to* j.

Pseudocode: Floyd-Warshall

```
Metagraph(G(V, E)):
  for i ∈ V do
    for j ∈ V do
      d(i, j, 0) ← ℓ(i, j)
        (* ℓ(i, j) ← ∞ if (i, j) ∉ E, 0 if i = j *)

  for k ← 0 to n − 1 do
    for i ∈ V do
      for j ∈ V do
        d(i, j, k) ← min { d(i, j, k − 1),
                           d(i, k, k − 1) + d(k, j, k − 1) }
  for v ∈ V do
    if d(i, i, n − 1) < 0 then
      return "∃ negative cycle in G"

  return d(·, ·, n − 1)
```

Running time: $\Theta(n^3)$