COMP3121 Exam Notes by Hae Kim

Divide and Conquer

Solution Framework

1. Establish **divide**; your splitting. How many subproblems? What are they?
2. Establish **conquer**; what is your recursion? Are you doing anything in your recursion? What's the order of your recursion?
3. Establish **combine**; what are you doing after your recursion? Why is it needed?
4. Establish correctness and time complexity.

Justification

Time

Time complexity justification can be done using the Master Theorem. For some recurrence:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

1. If $f(n) = O(n^{log_b a - \epsilon})$: $\epsilon \in R$, then $T(n) = O(n^{log_b a})$
2. If $f(n) = O(n^{log_b a} log^k n)$: $k \geq 0$ then $T(n) = O(n^{log_b a} log^{k+1} n)$
3. If $f(n) = \Omega(n^{log_b a + \epsilon})$: $\epsilon > 0$ and for some k < 1 and $af(n/b) \leq kf(n)$, then $T(n) = \Theta(f(n))$

Correctness

Divide and conquer is by nature, an inductive algorithm. Hence, we can prove everything by induction. Specifically:

- The inductive hypothesis is the conquer step.
- The combine step is the inductive step.
- The divide step is proven *by the combine step.*

Thus, we assume that for the size of our subproblems, our algorithm is correct – then prove that at the next combine, we still maintain a correct answer, validating our divide.

Lecture Algorithms

Karatsuba Algorithm

Given an integer a_0 in bit-form, we can represent the integer, by it's two halves $a_1 \cdot 2^{n/2} + a_0$. Thus, we can represent the multiplication of two integers as:

$$a_1 b_1 2^n + \big((a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0\big)2^{n/2} + a_0 b_0$$

Thus, we save one multiplication – leading to a total time complexity of $\Theta(n^{1.58})$.

Fast Fourier Transform

The Fast Fourier Transform (FFT) is an algorithm that runs the Discrete Fourier Transform in an efficient, $O(n \log n)$ time.

The FFT evaluates $2n + 1$ points on a function using the properties of the roots of unity – whereby since unities are mirrored, at each divide, only half the roots must be calculated (a la fast exponentiation).

The FFT can be used in questions beyond polynomial multiplication, such as to find all the values of a cartesian product across two arrays.

Strassen Algorithm

The Strassen Algorithm reduces the usual 8 multiplications that is required in matrix multiplications to 7, through a clever use of repeated multiplications.

The algorithm divides subproblems into four matrices, by dividing the width of the matrices by two. Thus, we get the recurrence:

$$T(n) = 7T(n/2) + O(n^2)$$

where $T(n) = n^{2.81}$. The driving function has the addition and subtraction of matrices, and hence is quadratic.

Greedy

Solution Framework

1. Define your algorithm.
2. Define your greedy heuristic.
3. Prove by *exchange argument* or *greedy stays ahead*

Exchange Argument

The exchange argument works by considering some other solution **O**, which agrees with the greedy solution **G**, up till $k - 1$, and that at the k-th index, differ. We want to show that by switching $g_k \rightarrow o_k$, that we improve/do not change the optimality of **O**.

Greedy Stays Ahead

Greedy stays ahead uses an inductive argument. You show for the base case, that the greedy argument does return a correct optimal response. Then, assume the greedy argument is true until the $k - 1$-th element, and show that for the k-th element, the greedy solution is still optimal using the assumption.

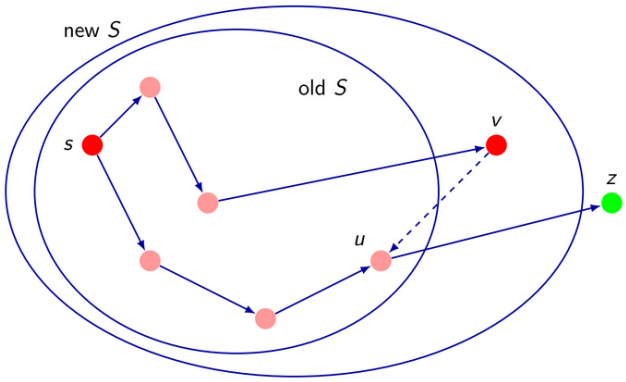
Heuristics

We can disprove heuristics using counter examples, and prove heuristics using the above two methods.

Graphs

Djikstra's Algorithm

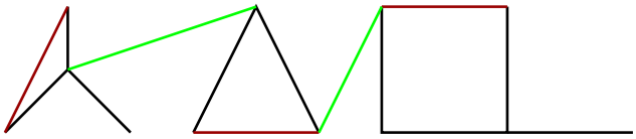
Djikstra's algorithm's optimality is show through contradiction, by assuming the possibility that by the changing of d_z, we could still have some penultimate vertex $\neq v$ for z.



The algorithm is implemented using an augmented heap, which is a heap that also stores the index of an element. We need this to efficiently update any element.

Kruskal's Algorithm

Kruskal's Algorithm is proven by showing that a set of subtrees can be connected with a minimal edge to create the MST.



The algorithm is implemented using the Union-Find data structure, which has three operations:

1. Initialise(S), which returns a structure in which all items are placed into distinct singleton sets.
2. Find(a), which returns the (label of the) set to which a belongs.
3. Union(a, b), which merges the sets A and B (witch labels a and b) into a single set A U B. The smaller set gets merged into the larger set under it's label.

Flow Networks

Solution Framework

1. Define your flow network, this include:
    a. Your source/super source and their connections; ensuring you declare the capacities of the edges from the source.
    b. Your intermediary vertices, their capacities (if they exist), and the connections that exist between them with their capacities.
    c. Your sink/super sink and their connections; ensuring you declare the capacities of the edges to the source.
2. Validate that your flow network reflects the constraints – mention constraints, exclusion of vertices, etc.
3. Apply a flow networks algorithm – most usually Edmonds Karp.

Residual Graph

A residual graph represents the remaining flow in the forward direction, and the used flow in the opposite direction for each edge.

If a path exists in the residual graph from the sink to the source – then we have yet not reached maximum flow.

Minimum Cut

The minimum cut represents the "bottleneck" of the flow network. They are a set of edges that minimally represent how much flow is within the system at any time. There can be more than one minimum cut.

A minimum cut can be found in $O(V + E)$ using your preferred graph traversal method.

Ford-Fulkerson & Edmonds-Karp

Both continually relax edges within the residual graph. Ford-Fulkerson chooses any path, whereas Edmonds-Karp chooses the shortest path.

This leads Edmonds-Karp to a tighter upper bound of $O(VE^2)$, however, the $O(E|f|)$ still holds from Edmonds-Karp. You choose whichever's tighter for the problem.

Maximum Bipartite Matching

A *matching* in a graph G is a subset $M \subseteq E$ such that each vertex of the graph belongs to *at most* on edge in M.

A *maximum bipartite matching* is a matching in G containing the largest number of edges where G is a bipartite graph. We can solve this problem using flow networks.

This is a common manipulation you are asked to do in a flow networks question as using the "maximum number of edges" could be re-framed into many real life applications.

Common manipulations

- Capacity = 1 to represent a path used/the number of edges used in a min cut. Very common.
- Excluding vertices/edges that do not follow a constraint.
- The use of infinite edges to always include a set of vertices within a min cut or a side of a min cut.

Dynamic Programming

Solution Framework

1. Variables (e.g $for\ 0 \leq i \leq n$)
2. Problem definition (e.g Let $P(i)$ be the problem of evaluating $opt(i)$)
3. Recurrence (be careful of re-adjusting bounds from original declaration)
4. Base cases
5. Order of computation
6. Final answer
7. Justification of correctness and time

Justification

Since dynamic programming is a brute force algorithm that is efficient, usually justifying that the recurrence satisfies every possible selection at a given move is enough.

Time is easy: $subproblems \times time\ for\ each\ subproblem$

Order of computation

To find the order of computation, see what (i, j, k) relies upon:

$$(i, j, k) = (i - 10, j - 4, k - 1)$$

Here, if we go in ascending order of k, then we are valid in our subproblems (if we have defined properly for base cases).

Bellman-Ford

Single source, shortest path. Where i represents the amount of edges uses, and t represents the current vertex.

$$opt(i, t) = min\{opt(i - 1, v) + w(v, t) \mid (v, t) \in E\}$$

Base case: $opt(0, s) = 0, opt(0, t) = 0$

Floyd-Warshall

All pairs, shortest path. For $1 \leq i, j \leq n$ and $0 \leq k \leq n$:

$$opt(i, j, k) = min\{opt(i, j, k - 1), opt(i, k, k - 1) + opt(k, j, k - 1)\}$$

Base case: 0 if $i = j$, the edge for an edge that exists in the edge set, and infinity where an edge doesn't exist between i and j.

Rabin-Karp

The Rabin-Karp algorithm utilises hashing to speed up average case performance.

$$H(A_{s+1}) = d \cdot H(A_s) - d^m a_s + a_{s+m}\ mod\ p$$

The hashing multiplying descending powers of the alphabet size to each alphabetical index of the characters.

Knuth-Morris-Pratt

Uses the failure function π, to use previous answers. While the brute force always starts from the beginning of the string, KMP re-uses prefix-suffixes within the string match. Runs in $O(n + m)$ time.

The failure function gives you the index of the previous prefix from your current sequence.

KMP as Finite Automata

k	matched	x	y	z
0		**1**	0	0
1	x	1	**2**	0
2	xy	**3**	0	0
3	xyx	1	**4**	0
4	xyxy	**5**	0	0
5	xyxyx	1	4	**6**
6	xyxyxz	**7**	0	0
7	xyxyxzx	1	2	0

The transition function of the finite automata representation represents for each letter at a specific state, which state it would go to next given the next letter.

For example, at state 5, if x is the next letter, we must restart from 1.

Intractability

Reductions Framework

NOTE: We are often asked to find decision problems regarding some inequality "at most" or "at least". Always try to solve for the extreme case – if we are asked to solve for "at most 2024", we want to solve for 2024.

1. Establish the forwards direction – $X \rightarrow Y$; denote some property of the YES instance that can be transposed to Y's problem constraints.
2. Establish the backwards direction – you can either prove that NO X → NO Y; or prove YES Y → YES X; whichever is easier.
3. Establish the reduction takes a polynomial amount of time. *Always remember to take the length of the input in consideration.*

Time complexity, lengths of input

- We use binary numbers, so each integer takes $log_2 x$ space for some number x.
- A character takes a byte, so an n-length string has size $O(n)$
- An adjacency list takes $O(E \log W)$ space, whereas an adjacency matrix takes $O(V^2 \log W)$ space.

With these in mind – Big-Oh complexity should be measured with reference to the size of the input.

P, NP, NP-Hard, NP-Complete

- **P** is a problem class where we know how to solve it in polynomial time.
- **NP** is a problem class where we know how to verify it in polynomial time.
- **NP-Hard** is a problem class where all NP problems can reduce to it.
- **NP-Complete** is NP-Hard ∩ NP.

Polynomial Reductions

A polynomial reduction from some problem X to some problem Y, takes some function $f(x)$ such that:

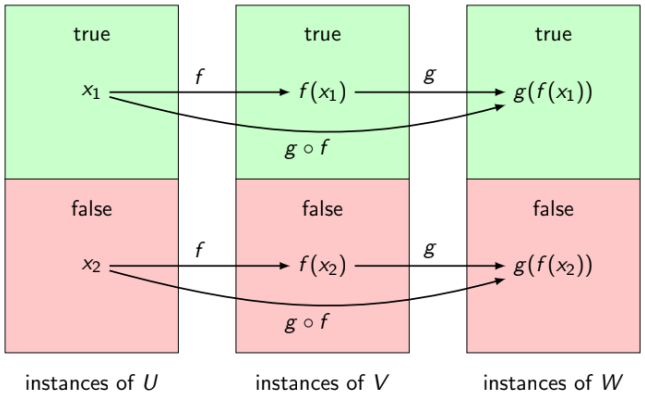
1. $f(x)$ maps instances of X to instances of Y.
2. f maps YES instances of X to YES instances of Y and NO instances of X to NO instances of Y (or YES instances of Y to YES instances of X).
3. $f(x)$ is computable in polynomial time.

A reduction need not be surjective. A reduction from $X \rightarrow Y$ suggests that:

1. X is no harder than Y, as instances of X can solve instances of Y.
2. Whatever problem class X is in Y must also be in.

NP-Completeness Proof

Let V be an NP-Complete problem, and let W be another NP problem. If V is polynomial reducible to W, then W is also NP-Complete. We use another NP problem U, to satisfy this proof.



Linear Programming and super-polynomial algorithms

LP Solutions Framework

1. Establish the variables that exist within your linear programming.
2. Establish the constraints that exist with respects to your variables.
3. Establish the function you are trying to optimise.
4. State a LP algorithm that is appropriate with your problem context.

Super-polynomial Solutions Framework

The same as any other algorithms solution, but very likely you will have some subset type expression like 2^n, as they are often exhaustive brute forces.

Some can be improved using dynamic programming; but are still super-polynomial.

Integer vs Reals

Integer linear programming is NP-Hard, because there exists so many more corners on the convex polygon generates by the constraints – as they cannot lie on any real number.

Real numbers have a guaranteed polynomial time algorithm, called the ellipsoid algorithm. However, the *SIMPLEX* algorithm is much more commonly used.