Contents

1 Divide and Conquer

Divide and conquer is one of the most important concepts in algorithms! If you forget almost everything else in this class, don't forget this one. It will help you if you are ever the general of an army too.

The general idea is:

1. **Divide:** Divide the problem into subproblems
2. **Conquer:** Solve each subproblem either by dividing it further or, if it is simple enough, solving it.
3. **Combine:** Combine the solutions to the subproblems into the actual answer.

Example: Merge Sort on a sorted array A

1. **Divide** the array into two halves
2. **Conquer** each half by dividing it again or returning it if it has only one element
3. **Combine** the two halves together in a merge.

2 Master Theorem

This is a method for solving divide and conquer problems. Specifically given a recurrence relation

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

we think about the recursion tree (see Lecture 8). This gives us three cases:

1. $f(n) = O(n^{\log_b(a)-\epsilon}) \Rightarrow T(n) = O(n^{\log_b(a)})$ Example: $T(n) = 8T(n/2) + O(n)$
2. $f(n) = O(n^{\log_b(a)} \log^k(n)) \Rightarrow T(n) = \log(n) f(n)$ Example: $T(n) = 2T(n/2) + O(n)$
3. $f(n) = O(n^{\log_b(a)+\epsilon}) \Rightarrow T(n) = f(n)$ Example: $T(n) = 2T(n/2) + O(n^3)$

Figure 1: DRAW AVL ROTATIONS HERE!

3 Binary Search Trees

Nodes in a binary search tree have two properties:

1. Nodes have 0, 1, or 2 children
2. All nodes in the left subtree of node p are less than or equal to p and all nodes in the right subtree of p are greather than or equal to p.

Some vocabulary:

1. *Height*: Leaves have height 0. The height $h(n)$ of a non-leaf node n is defined as $h(n) = \max(h(n_l), h(n_r))$ where n_l and n_r are the left and right children of n respectively.
2. *Successor*: The sucessor of a node is the node in the tree with the next-largest value.
3. *Predecessor*: The predecessor of a node is the node in the tree with the next-smallest value.

3.1 AVL Trees

AVL Property: For all nodes n, $h(n_l)$ and $h(n_r)$ differ by at most 1.

We maintain this property using rotations as shown in Figure 1.

Running Times: We discussed the following AVL algorithms. You should know how they work and their running times:

- Insert: $O(\log n)$
- Delete: $O(\log n)$
- One rotation: $O(1)$
- Successor/Predecessor: $O(\log n)$

3.2 Augmenting Binary Search Trees

We can keep “extra” information with nodes *provided* we can keep this information correct while still inserting/deleting nodes in logarithmic time.

Rank and Select: By augmenting a node with the number of children in its left subtree and the number of children in its right subtree we can solve two common problems in logarithmic time:

Rank: Given a node n, what is its position in a sorted list of nodes?

Select: Given a position r in the sorted list of nodes, what node is at this position?

Problem: Problem 2d from Problem Set 2: Find the smallest node larger than t such that the node’s successor is at least 6 greater.

Solution: Store the number of gaps at least 6 in left and right subtrees. Now find the successor of t in the tree. If this successor has a gap, return it. Otherwise, if it has a gap in its right tree, recursively search for the smallest number with a gap in the right tree. Otherwise, find the first node n such that n is a left child and has a gap or has a gap in its right subtree. If n has a gap in its right subtree, search that subtree for the smallest node with a gap.

Why does this work? Let the x-successor of t be the node that is x nodes larger than t. So the 1-successor is the successor of t and the 2-successor is the successor of t’s successor etc. We want to find the smallest x such that the x-successor of t has a gap. Now the successor of a node is in its right subtree if it has one or one of its ancestors, specifically the first ancestor that is a left child of its parent, if it does not. Therefore, if a node n does not have a gap in its right subtree, the successor of the largest node in its right subtree is the first ancestor of n that is a left child. Therefore, we check all x-successors until we find the first one with a gap.

4 Sorting

4.1 Comparison Sort

Any algorithms that depend on comparing elements for sorting are *comparison sort algorithms.* Such as:

- Insertion sort: $O(n^2)$
- Merge sort: $O(n \log n)$
- Heap sort: $O(n \log n)$

Lower Bound: There is an $O(n \log n)$ lower bound on comparison sorts.

4.2 Counting and Radix Sorts

If we can map all elements to integers from 1 to C then we can use *counting sort* which runs in time $O(n + C)$.

4.2.1 Counting Sort (IF TIME)

Counting sort can beat comparison sort bound *because* it doesn't use comparisons. In order not to use comparisons, we must have a little bit of "extra" knowledge. Namely: given an array A to sort, we need to be able to map every element that might appear in A uniquely to integers in $[1, k]$ where k is a small integer.

Input: A array to be sorted

Output: B sorted array of elements of A

Pass 1: Create array C of length k. $C[i]$ stores the number of times i appears in A. For each i, $C[A[i]] = C[A[i]] + 1$

Pass 2: For each entry i in C, put $C[i]$ i's in B.

This takes $O(n + k)$.

Problem: B is not *stably* sorted. Two equal keys may swap their relative orders. We would like to avoid that.

New algorithm: Create C', which stores in $C'[i]$ number of numbers in A less than or equal to i. Fill in C' after filling in C just by keeping running total.

Now, for $j = \mathit{length}[A]$ downto 1, place $A[j]$ at $C'[A[j]]$ in B and decrement $C'[A[j]]$. Now the sorting is stable.

Example: We know we are sorting numbers from 1 to 8

$$\begin{aligned} A &= [2, 7, 5, 3, 5, 4] \\ C &= [0, 1, 1, 1, 2, 0, 1, 0] \\ C' &= [0, 1, 2, 3, 5, 5, 6, 6] \end{aligned}$$

Stepping through the second pass (0 in B indicates nothing there yet):

1. $$\begin{aligned} B &= [0, 0, 0, 0, 0, 0] \\ C' &= [0, 1, 2, 3, 5, 5, 6, 6] \end{aligned}$$

2. $$\begin{aligned} B &= [0, 0, 4, 0, 0, 0] \\ C' &= [0, 1, 2, 2, 5, 5, 6, 6] \end{aligned}$$

3. $$\begin{aligned} B &= [0, 0, 4, 0, 5, 0] \\ C' &= [0, 1, 2, 2, 4, 5, 6, 6] \end{aligned}$$

4. $$\begin{aligned} B &= [0, 3, 4, 0, 5, 0] \\ C' &= [0, 1, 1, 2, 4, 5, 6, 6] \end{aligned}$$

5.

$$\begin{aligned} B &= [0,3,4,5,5,0] \\ C' &= [0,1,1,2,3,5,6,6] \end{aligned}$$

6.

$$\begin{aligned} B &= [0,3,4,5,5,7] \\ C' &= [0,1,1,2,3,5,5,6] \end{aligned}$$

7.

$$\begin{aligned} B &= [2,3,4,5,5,7] \\ C' &= [0,0,1,2,3,5,5,6] \end{aligned}$$

4.2.2 Radix Sort

Digit-by-digit sort of list of numbers. Sort on least-significant digit first using a *stable* sort.

DON'T DO BELOW

Why least significant? Example: Most significant

$$\begin{array}{ccccc} 33 & & 33 & & 52 \\ 55 & \rightarrow & 55 & \rightarrow & 33 \\ 52 & & 52 & & 55 \end{array}$$

Oops!

Why do we need a stable sort? Because we don't want to mess up orderings caused by earlier digits in later digits!

Example: Sort 33, 55, 52

$$\begin{array}{ccccc} 33 & & 52 & & 33 \\ 55 & \rightarrow & 33 & \rightarrow & 52 \\ 52 & & 55 & & 55 \end{array}$$

We can only guarantee that 52 comes before 55 if we can guarantee the sort on the second digit is stable!

EXPLAIN RUNNING TIME!

Running Time: Sorting n words of b bits each: Each word has b/r 2^r-base digits. Example: 32-bit word is has 4 8-bit digits. Each counting sort takes $O(n + 2^r)$, we do b/r sorts. Choose $r = \log n$, gives running time $O(nb/\log n + n)$. *Radix sort is not sublinear in n.*

For many applications, you do not need to worry about bits. If you know you have C digits, where C is constant, then you do C sorts of $O(n)$ for a running time of $O(Cn) = O(n)$.

5 Hashing

Hashing is a way of checking membership in sets in constant time. We create a table of size m and a function h that maps all possible elements k to $1...m$.

The only problem is that it may not be possible to map all elements to a different entry in the table. We discussed two ways to deal with *collisions*:

- Chaining: At each entry of the table, we create a linked list of all entries that map there.
- Probing: Our hash function h is a function of k *and* i the number of collisions we have seen so far. We looked at a linear probe: $h(k, i) = h'(k) + i$ where $h'(k)$ is a hash function. We also discussed open addressing: $h(k, i) = (f(k) + ig(k)) \bmod m$ where f and g are hash functions.

In order to keep our constant time lookups and deletes, we discussed two assumptions we need on our hash functions:

- Simple Uniform Hashing Assumption (SUHA): Every key k is equally likely to land in any bucket and this is independent of where any other key lands. This is useful to keep chaining manageable.
- Uniform Hashing Assumption (UHA): For a key k, the probe sequence is a random permutation of $1...m$. This is useful to keep probing manageable.

Problem: Hashing Probabilities We insert four keys into a hash table that uses linear probing. Assume simple uniform hashing. What is the probability that inserting the fourth key requires three probes?

Solution: This requires the first three keys to be next to each other. The cases for where the first are:

1. k_1 hashes anywhere (1), k_2 hashes 1 below k_1 ($2/m$), k_3 hashes below k_2 ($3/m$): $6/m^2$
2. k_1 hashes anywhere (1), k_2 hashes 1 below k_1 ($2/m$), k_3 hashes above k_1 ($1/m$): $2/m^2$
3. k_1 hashes anywhere (1), k_2 hashes 2 below k_1 ($1/m$), k_3 hashes below k_1 ($2/m$): $2/m^2$
4. k_1 hashes anywhere (1), k_2 hashes 1 above k_1 ($1/m$), k_3 hashes below k_1 ($2/m$): $2/m^2$
5. k_1 hashes anywhere (1), k_2 hashes 1 above k_1 ($1/m$), k_3 hashes above k_2 ($1/m$): $1/m^2$
6. k_1 hashes anywhere (1), k_2 hashes 2 above k_1 ($1/m$), k_3 hashes to 1 above k_1 ($1/m$): $1/m^2$

One of these three cases must occur ($14/m^2$) and the last key must collide with the top key ($1/m$) for a total probability of $14/m^3$.

6 Heaps

Max-Heap Property: For any node, the keys of its children are less than or equal to its key.

We discussed the following heap algorithms. You should know them and their running times:

- Heapify: Take an array where the left and right children of the root are heaps, but the array itself might not be and return an array. $O(\log n)$.
- Build-Heap: Build a heap from an unsorted array. $O(n)$
- Increase-Key: Increase the value of one key. $O(\log n)$
- Sort: Sort an unsorted array using a heap. $O(n \log n)$.
- Find-Max: Return the max element. $O(1)$
- Extract-Max: Remove and return the max element. $O(\log n)$
- Insert: Insert a key. $O(\log n)$

7 Shortest Paths

7.1 All Pairs Shortest Paths

The APSP problem is finding the shortest distance between all pairs of points. We discussed several algorithms for this (see the dynamic programming section for details):

- Repeated Bellman-Ford: $O(|E||V|^2)$
- Repeated Dijkstra: If all weights are positive, we can also run Dijkstra's algorithm from each vertex for a running time of $O(|V||E| + |V|^2 \log |V|)$.
- APSP-Slow: $O(|V|^4)$
- APSP-Fast: $O(|V|^3 \log |V|)$
- Floyd-Warshall: $O(|V|^3)$

7.2 Constrained Shortest Paths and Graph Transformations

We often want to find constrained shortest paths. In many cases, Dijkstra's algorithm or Bellman-Ford work for this. For example

Problem: Problem 4 Fall 2008 Final (modified): Consider a directed graph $G = (V, E)$ where each *edge* $(u, v) \in E$ is either red or blue. The weight of a path is the sum of the weights of the edges on this path *plus* 5 for each pair of adjacent edges that are different colors. Assume all weights are non-negative and give an efficient algorithm for computing the cost of the shortest path from s to t in G.

Solution: Construct a new graph $G' = (V', E')$. For each $v \in V$, add v_R and v_B to V'. For each edge $(u, v) \in E$ create the edge (u_R, v_R) in E' if (u, v) was red and (u_B, v_B) if (u, v) was blue. Finally, for each vertex $v \in V$, create edges (v_B, v_R) and (v_R, v_B) each with a cost of 5 in E'. Run Dijkstra from s_B and s_R and return $\min(\delta(s_R, t_R), \delta(s_R, t_B), \delta(s_B, t_R), \delta(s_R, t_B))$.

This works becaues if we wish to switch edge colors, we must traverse an edge of weight 5.

7.3 Vertex Potentials

If we know where we're going and we have non-negative weights, we can use vertex potentials to help push us there! A potential function is an "estimate" of the distance to a target vertex t. For target vertex t the potential function $\lambda_t(v)$ is calculated for every vertex v. We modify the weights using

$$w^*(u,v) \leftarrow w(u,v) - \lambda_t(u) + \lambda_t(v)$$

Since we then want to run Dijkstra's algorithm on the modified weights, we must have $w^*(u,v) \geq 0$ for all $u, v \in V$. A potential function for which this is true is a *feasible potential*. We discussed two potentials:

- **Landmark**: Choose a landmark l (train station, airport, central vertex, whatever). Precompute $\delta(v,l)$, the shortest path from v to l for all $v \in V$. The potential function is

$$\lambda_t^{(l)}(v) = \delta(v,l) - \delta(t,l)$$

- **Euclidean distance**: If weights have some correspondence to Euclidean distance, we can use the potential function

$$\lambda_t(v) = \frac{l(v,t)}{v_{\max}}$$

  where $l(u,v)$ is the Euclidean distance from u to v and $v_{\max} = \max_{u,v\in V}(l(u,v)/w(u,v))$.

You can prove both of these potentials are feasible (see Recitation 17).

8 Dynamic Programming

8.1 Strategies for DP Problems

Optimal substructure is key to dynamic programming and the key to dynamic programming problems is figuring out what the optimal substructure *is*. Usually, once you know that, you can solve the problem!

By optimal substructure, we mean a set of problems $p_0, ..., p_n$ such that each problem $p_i = f(p_0, ..., p_{i-1})$ is a function only of the problems below it. These are the "sub-problems"; we are usually interested in only the answer to p_n, but must solve $p_0, ..., p_{n-1}$ in order to find p_n.

Here are some strategies for identifying the correct sub-problem:

- Figure out what you can recurse or index over. Sub-problems are indexed so chances are you have something else with indices in the problem. For example, if you have a sequence $a_1, ..., a_n$, trying the ith sub-problem being the answer for the sequence from 1 to i is a good place to start. If you have a graph, you probably want to iterate over nodes and/or edges. If you have bags of gold, you probably want to iterate over those.
- If there are integers in the problem upper bounded by some integer M, you probably need to index over every integer from 1 to M. See the Making Change and Balanced Partition problems for examples.
- If it is a decision problem, you probably want to iterate over a 0/1 matrix where 0/1 represents the answer to the decision problem rather than a number.

- Once you have some idea what the sub-problems might look like, *start at the top*. Assume you know the answers to problems $p_0, ..., p_{n-1}$ and figure out how you use those to find p_n. Try for a little while. If:
  - It worked! Analyze the running time and you are done.
  - I can't figure out what p_n should be! Chances are, you didn't choose the correct set of sub-problems. Try adding an index. For example, if you had a sequence and you were trying to figure out the solution for elements from 1 to i, now try to figure out the solution for elements from i to j.

8.2 Running Time of DP

If you've written down a good recursion, figuring out the running time is easy. Read two things off your recursion relation:

1. The *number of sub-problems.* Your recursion is over something like $A(i, j, k, l, ...)$. Each entry of A is a different sub-problem so just calculate the size of A. For example, if A has two indices and the first goes up to n and the second goes up to m then A is size mn.
2. The *running time of each sub-problem.* Look at your recursion relation over quantity A. Assume you have already solved all needed subproblems so any entry of A that you require is filled in. Figure out the running time of the recursion function with that assumption. For example, if the recursion requires comparing two different values in A, it is constant time since you assume you know both of those values already. If it requires comparing n values, it is $O(n)$ time.

Multiply the number of sub-problems by the time it takes to solve each one. That's it!

8.3 Top-Down vs Bottom-Up

We discussed two ways of thinking about DP. The first is top-down. Given the recursive relationship $p_j = f(p_{j-1}, ..., p_0)$, create f and memoize it. This is the easiest way of programming dynamic programming, but, although it will be assymptotically the same performance, it is not always the most efficient algorithm in terms of constants. It's also tricky to figure out the running time.

Therefore, we also think about bottom-up dynamic programming, in which we first solve p_0 then p_1 on up. This generally requires a little care in the indexes of the for loop, but makes it easier to evaluate the running time.

8.4 All Pairs Shortest Path

Many of the algorithms we discussed for APSP were DP algorithms. We used the following recurrence relations:

APSP-Slow: Let $d_{ij}^{(m)}$ be the shortest path that uses at most m edges:

$$d_{ij}^{(m>0)} = \min_k \left(d_{ik}^{(m-1)} + w(k, j) \right)$$

d is size $O(|V|^3)$, each evaluation requires $O(|V|)$ for running time $O(|V|^3)$.

APSP-Fast: Same memo, but use

$$d_{ij}^{(2m)} = \min_k \left(d_{ik}^{(m)} + d_{kj}^{(m)} \right)$$

Now we only fill in $O(|V|^2 \log |V|)$ elements of d, each taking $O(|V|)$ time for $O(|V|^3 \log |V|)$ time.

Floyd-Warshall: Let $d_{ij}^{(k)}$ be the shortest path from i to k using only vertices $v_1, ..., v_k$:

$$d_{ij}^{(k)} = \begin{cases} w(i,j) & \text{if } k = 0 \\ \min \left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right) & \end{cases} \tag{1}$$

d os size $O(|V|^3)$, each evaluation requires $O(1)$ for running time $O(|V|^3)$.

8.5 Examples

A large number of example problems and their solutions have been posted on the class website. The ones we did in the review session were:

Problem: Problem 4 Spring 2010 Final: Let T be a tree with edge weights $w(e)$. The maximum weight matching (MWM) in T is a subset of edges of maximum total weight such that no two edges are incident on the same vertex. Find an MWM in T.

Solution: For each vertex, we will either include 0 or 1 of its incident edges. Therefore, we iterate over M, the MWM assuming v is the root of the tree. We also keep C, the MWM among the children of v.

$$\begin{aligned} M(v) &= \begin{cases} 0 & \text{if } v \text{ is a leaf} \\ \max_{\text{children } c}(w(v,c) + C(c), M(c)) & \text{else} \end{cases} \\ C(v) &= \begin{cases} 0 & \text{if } v \text{ is a leaf} \\ \max_{\text{children } c}(M(c)) & \text{else} \end{cases} \end{aligned}$$

We return $M(root)$. In evaluating C and M, we do one operation for every edge in the tree. Since this is a tree, the number of edges is $O(n)$ for a total running time of $O(n)$.

9 Numerics and Cryptography

9.1 Things to Know

- Why numerics is important
- Heron's and Newton's methods, when they converge, how fast they converge
- Lower bounds for multiplying and adding two numbers
- Public key/private key cryptosystems exist and what they are
- If the input to an algorithm is a number n, the size of the input is $\log(n)$ so "polynomial time algorithms" run in time $\log(n)$.
- Modular exponentiation

9.2 Modular Exponentiation

Another thing that crops up all over cryptography is modular exponentiation. We would like to be able to do that quickly.

```
ExponMod($c, d = \langle d_{k-1}, d_{k-2}, ..., d_0 \rangle, n$)
    Input: Base $c$, exponent $d$ of $k$ bits, modulus $n$
    Output: $c^d \bmod n$.
1   $i \leftarrow 0, m \leftarrow 1$
2   for $j = k - 1$ downto $0$
3       $i \leftarrow 2i$
4       $m \leftarrow m^2 \bmod n$
5       if $d_j = 1$
6           $i \leftarrow i + 1$
7           $m \leftarrow mc \bmod n$
8   return $m$
```

Running Time: If n, c, and d are k-bit numbers, total bit operations is $O(k^3)$. (Assuming multiplication of k-bit numbers takes $O(k^2)$.)

Correctness: Firstly note that we output $m = c^i$. Therefore, we we must just show that $i = d$. To do this, note that multiplying by 2 in binary is equivalent to just adding a zero to the end of the number. Then proceed by induction by decreasing bit significance to show that the bits of i and d are the same:

Base Case: If $d_{k-1} = 0$ then $i = 0$ after the first iteration of the for loop since the if statement is not entered. If $d_{k-1} = 1$ then $i = 1$ after the first iteration since the for loop is entered.

Induction Step: Assume bits $k - 1, ..., l + 1$ are correct. Then for bit l, if $d_l = 0$, $i_l = 0$ since we double i but do not add 1 and if $d_l = 1$, $i_l = 1$ since we double i and add zero.

10 NP-Complete and Reductions

10.1 Direction of Reductions and Hardness of Problems

When we say "reduce problem A to problem B", we mean show that if you can solve problem B in polynomial time, you can solve problem A in polynomial time. To do this, assume you have a *black-box* algorithm for B. A black-box algorithm is one you can use, but don't know how it works and cannot change. Show that you can use this black-box algorithm to solve A in time polynomial in the input to A and T where T is the running time of the black-box algorithm.

If A can be reduced to B in polynomial time then, if we can solve B, we can solve A. Therefore, B is *at least as hard as A*. B might be harder than A unless we can show the reduction the other way around, but certainly it is at least as hard.

Specifically, if A is an NP-Hard or NP-Complete problem and we can use B to solve A, B must also be NP-Hard or NP-Complete.

Example: Factoring is as hard as RSA: We reduce RSA to factoring: If we could factor, we could find p and q for RSA, which allows us to find $\phi(n) = (p-1)(q-1)$, which allows us to find $d = e^{-1} \bmod \phi(n)$. Therefore, factoring is *at least as hard as RSA*. However the reverse has never been proven to be true! Breaking RSA is *not* known to be as hard as factoring. No one has found a reduction from factoring to RSA.

Decision Problems: Reductions do *not* have to be between decision problems! We discussed that type of reduction in class because decision problems are an important class of problems. However, as above, you can reduce between any two types of problems so long as you can translate the output of one to the output of the other.

Polynomial Time Reductions: Usually, we get sloppy with our notation and just ask for a *reduction*. Implicit in this is that the reduction must actually be a *polynomial time reduction*. If the reduction itself takes exponential time it tells us nothing!

10.2 Examples

Example problems are up on the class website. In the review session we also did:

Problem: Partition and Two-Machine Scheduling: Consider the following two problems:

- Partition: Given a set n of non-negative integers $\{a_1, ..., a_n\}$, decide if there is there a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$.
- TwoMachineSchedule: You must schedule n tasks on two machines. Both machines have the same speed, each task can be executed on each machine, and there are no restrictions on which tasks must be run on which machine. You are given executation times $a_1, ..., a_n$ and a deadline D. Can you complete the tasks within the deadline?

Reduction of Partition *to* TwoMachineSchedule: We first check if $S = \frac{1}{2}\sum_{i=1}^{n} a_i$ is an integer. If not, return FALSE. Otherwise, assume we have a black-box algorithm for TwoMachineSchedule and return this algorithm run on $a_1, ..., a_n$ with deadline S. This works since if we partition $a_1, ..., a_n$ into P_1 and P_2, $\max(\sum_{a_i \in P_1} a_i, \sum_{a_i \in P_2} a_i) \geq S$.

Reduction of TwoMachineSchedule *to* Partition: Assume we are given an instance of TwoMachineSchedule, $a_1, ..., a_n$ and D and a black-box algorithm for Partition. Consider $I = 2D - \sum_{i=1}^{n} a_i$. Intuitively, I is the total idle time in any legal schedule. Note that if I is negative, the schedule is clearly impossible and we can just return FALSE.

Now we just need a set of integers $i_1, ..., i_m$ such that the set sums to I and we can make any number between 0 and I with this set. Then we could just return Partition on $a_1, ..., a_n, i_1, ..., i_m$. So what is $i_1, ..., i_n$? It is *not* I copies of 1! This would require I new numbers and I can be exponential in the size of the input space. So, although that would be a reduction, it would not be a *polynomial time reduction*. Instead, we use every power of 2 less than I, $i_1, ..., i_{m-1}$ and $i_m = I - \sum_{j=1}^{m-1} i_j$. This requires $O(\log I)$ numbers so is a polynomial time reduction.

11 Quick Proofs and How to Write Them

In general correctness proofs fall into one of three categories:

1. **Induction/Loop Invariants:** If you have a loop as the main part of your algorithm this is almost always the way to go! Do the induction over the index of the loop.
2. **If and only if:** If your algorithm is clearly returning one quantity q and you want to show this is equal to the desired quantity p. For example, if your algorithm returns TRUE iff there is a back edge and you want to show that this is equivalent to the algorithm returning TRUE iff there is a cycle. This is also the type of proof you almost always want to prove the correctness of a reduction since reductions are showing that being able to find an answer to one problem is equivalent to being able to find the answer to another.
3. **Contradiction:** When your algorithm is returning one quantity q and you want to show this happens whenever property p holds (but maybe not vice-versa). Also good for proving lower bounds on running times especially when doing reductions.

12 Extra Problems

BBST Problem: Select Explain how to do select in a balanced binary search tree logarithmic time.

Solution: Assume we traverse the tree down to a node n. We keep a counter t, initialized to 0, and each time we move right at node n we update $t \leftarrow t + 1 + L(n)$ where $L(n)$ is the number of nodes in the left subtree of n. By doing this, at any node n, t is the rank of the node. Therefore, when we encounter a node with $t > r$, we move left and if we encounter a node with $t < r$ we move right.

Dijkstra Problem: Problem 4 Fall 2010 Quiz 2 Consider a directed, weighted graph $G = (V, E)$ and $W \subseteq V$. We require that all paths pass through a vertex in W at least every 3 edges. Assume all weights are non-negative and give an efficient algorithm for finding the shortest path from s to every vertex in G.

Solution: Create a new graph $G' = (V', E')$. For every $w \in W$, add w to V'. For $v \in V \setminus W$, add 4 copies of v, v_1, v_2, v_3. For edges $(u, v) \in E$, with $u, v \in W$, add edge (u, v) to E'. For edges $(u, v) \in E$ with $u \in W$ and $v \notin W$, add edge (u, v_1) to E'. For edges $(u, v) \in E$ with $v \in W$ and $u \notin W$, add edges (u_1, v), (u_2, v), and (u_3, v) to E'. For edges (u, v) with $u, v \notin W$, add edges (u_1, v_2), and (u_2, v_3) to E'. Run Dijkstra etc.

Heap Problem: Problem 1b Fall 2010 Problem Set 3 in $O(n + i \log i)$ time Given an array of n numbers, find the largest i elements in sorted order.

Solution: Create a max-heap of the n elements in $O(n)$ time. However, we will not extract from this heap (which algorithm gave the $O(n + i \log n)$ running time given in the solutions), but auxilary heap in which we store candidates for the next maximum. Note that if we have extracted elements $x_1, x_2, ..., x_j$, the jth highest element must be a child (not descendent, actual child) of one of the $x_1, x_2, ..., x_j$. Therefore, in our auxilary heap, we store each node with pointers to its children *in the original heap.* We initialize the auxilary heap with the maximum element and extract maxes from the auxilary heap. When we extract a max from the auxilary heap, we add its children to the heap. This requires one extract-max and two inserts into the heap for each of the i elements, but the heap never grows beyond size $2i$ (since an element has at most two children) so each of these takes only $O(\log i)$. This gives us a total running time of $O(n + i \log i)$.

DP Problem: Problem 5 Fall 2007 Quiz 1 Given a sequence of integers $a_1, ..., a_n$ find a subset of $a_1, ..., a_n$ that maximizes the total sum subject to the constraint that you cannot select two adjacent elements.

Solution: Let $S(i)$ be the maximum sum of $a_1, ..., a_i$. The recursion is

$$S(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max(0, a_1) & \text{if } i = 1 \\ \max(S(i-2) + a_i, S(i-1)) & \text{else} \end{cases} \tag{2}$$

Return $S(n)$. There are $O(n)$ sub-problems each of which take $O(1)$ time for $O(n)$ total time.

DP Problem: Problem 3b Fall 2007 Final You have an n-by-n grid where each square (i, j) contains $c(i, j)$ gold coins. Assume $c(i, j) \geq 0$ for all squares. You start in the upper-left corner and end in the lower right corner and at each step you can only travel on square down or right. You collect all coins on the square you visit. Given an algorithm for finding the maximum number of coins you can collect.

Solution: Let $M(i, j)$ be the maximum number of coins possible to collect on an $i \times j$ board with (i, j) the lower right corner:

$$M(i,j) = \begin{cases} c(0,0) & \text{if } i, j = 0 \\ M(0, j-1) & \text{if } i = 0, j > 0 \\ M(i-1, 0) & \text{if } i > 0, j = 0 \\ \max(M(i-1, j), M(i, j-1)) & \text{else} \end{cases} \tag{3}$$

Return $M(n-1, n-1)$. There are n^2 subproblems, each of which take $O(1)$ time for total $O(n^2)$ running time.

Heap Problem: Problem 6 Spring 2010 final Assume you have an array where each element is no more than k from its sorted index. Sort the array.

Solution: Create a min-heap and initialize it to be the first k elements of the array. This must contain the actual smallest element since it can be no more than k from index 1. Extract the minimum element from the heap and add the next element in the array to the heap. The heap is always no more than size k and we perform n extractions for running time $O(n \log k)$.

Divide and Conquer Solution: Use merge sort. Except when merging the lists it is only necessary to merge the