Chris Umans *January 26, 2000*
Office Hours: 3:00–5:00 Tuesdays and Thursdays in Soda 583
`umans@cs.berkeley.edu`

Sections: 10:00–11:00 in Cory 289
11:00–12:00 in Etcheverry 3109

- **Graph representation practice:** If A is the adjacency matrix of a graph G, what does the transpose A^T represent? If G is an *undirected* graph this is just G. If G is a *directed* graph, A^T represents G with all of the edge directions reversed.

- **Graph representation practice:** If A is the adjacency matrix of an undirected graph G, what do the entries of AA^T represent? The diagonal entries (i, i) are the degree of vertex i. The other entries (i, j) are the number of paths of length exactly 2 from i to j.

- **Master Theorem, easier version:** This variant of the CLR Master Theorem is less detailed, but is sufficient for most situations:

  Theorem 1 *Given a recurrence $T(n) = aT(n/b) + \Theta(n^d)$, and $T(1) = \Theta(1)$, the solution is:*

  1. $\Theta\left(n^{\log_b a}\right)$, *if* $a > b^d$*;*
  2. $\Theta\left(n^d \log n\right)$, *if* $a = b^d$*;*
  3. $\Theta\left(n^d\right)$, *if* $a < b^d$*.*

- **Master Theorem practice:**

  - The running time of Mergesort obeys the following recurrence: $T(n) = 2T(n/2) + \Theta(n)$ and $T(1) = \Theta(1)$. Here $a = 2$, $b = 2$, and $d = 1$, so by the Master Theorem, we have $T(n) = \Theta(n \log n)$, since $a = b^d$.
  - The "naive" integer multiplication algorithm from section 6 of the reading yields the following recurrence: $T(n) = 4T(n/2) + \Theta(n)$, and $T(1) = \Theta(1)$. Here $a = 4$, $b = 2$, and $d = 1$, so by the Master Theorem we have $T(n) = \Theta(n^2)$, since $a > b^d$.
  - The improved integer multiplication algorithm from section 6 of the reading yields the recurrence: $T(n) = 3T(n/2) + \Theta(n)$, and $T(1) = \Theta(1)$. Here $a = 3$, $b = 2$, and $d = 1$, so by the Master Theorem we have $T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$, since $a > b^d$.

- **Homework 1, Problem 1.3-7, alternative solution:** The problem asks for a $\Theta(n \log n)$-time algorithm to determine if any two numbers out of a set S of n real numbers sum to a real number x. One idea is to sort the n numbers in increasing order $a_1, a_2, a_3, \ldots, a_n$, and then for each a_i, perform a binary search for $x - a_i$. If this binary search finds some other $a_j = x - a_i$, then we know that $a_i + a_j = x$, and we have found the desired two numbers. Otherwise, we know that for each a_i, there is no other a_j that sums to x. The sorting takes time $\Theta(n \log n)$, and we perform n binary searches, each taking time $\Theta(\log n)$, for a total of $\Theta(n \log n) + n \cdot \Theta(\log n) = \Theta(n \log n)$.

- **Homework 1, Problem 3-1(b), alternative solution:** We are asked for an asymptotically tight bound for:

$$S(n) = \sum_{k=1}^{n} \log^s k.$$

For the upper bound we notice that *each term* in the sum is at most $\log^s n$. There are n terms, so the $S(n) \leq n \log^s n = O(n \log^s n)$.

We'd like to use the same strategy for the lower bound, but the best *lower bound* we can give for each term is $\log^s 1 = 0$, which doesn't get us anywhere! Really, *most* terms in the sum are much larger than zero, and we can capture this intuition by "splitting the summation:"

$$S(n) = \sum_{k=1}^{n} \log^s k = \sum_{k=1}^{n/2-1} \log^s k + \sum_{k=n/2}^{n} \log^s k.$$

Now, if we use a lower bound on each term, the terms from the second half of the summation will give a meaningful lower bound:

$$S(n) = \sum_{k=1}^{n/2-1} \log^s k + \sum_{k=n/2}^{n} \log^s k \geq n/2 \cdot 0 + n/2 \cdot \log^s(n/2)$$

Remember that we want $S(n) \geq \Omega(n \log^s n)$, which means $S(n) \geq cn \log^s n$ for all $n \geq n_0$ for some constants c and n_0. We now notice that $\log(n/2) = \log n - 1$ is only "slightly" smaller than $\log n$. In fact we have $2(\log n - 1) \geq \log n$ for $n \geq 4$. Therefore, we have:

$$(2^{s+1})n/2 \cdot \log^s(n/2) = 2 \cdot n/2 \cdot (2(\log n - 1))^s \geq n \log^s n.$$

Since s is a constant, 2^{s+1} is also a constant, and finally, we have:

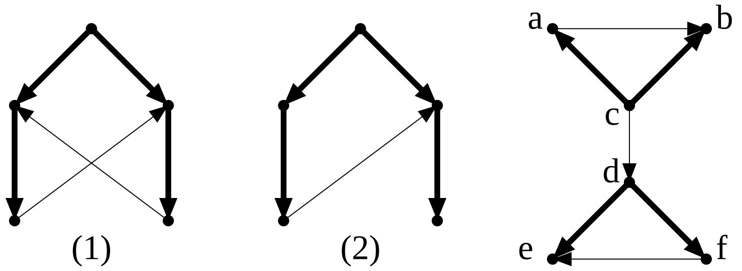
$$S(n) \geq n/2 \log^s(n/2) \geq \frac{1}{2^{s+1}} n \log^s n,$$

for $n \geq 4$, which implies $S(n) \geq \Omega(n \log^s n)$.

Chris Umans *February 2, 2000*
Office Hours: 3:00–5:00 Tuesdays and Thursdays in Soda 583
`umans@cs.berkeley.edu`

Sections: 10:00–11:00 in Cory 289
11:00–12:00 in Etcheverry 3109

- **DFS practice:** Given a graph G, is it true that every spanning forest of the graph is produced by the DFS algorithm for some ordering of the vertices and adjacency lists in the adjacency list structure? No! Some examples:



(1) (2) (3)

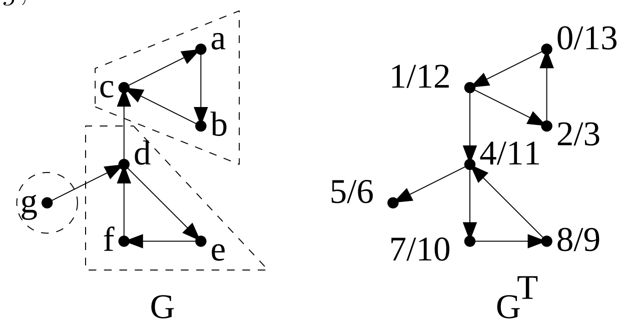
Example (1) cannot have been produced by a DFS. (Why?) However it could have been produced by a BFS. Example (2) could have been produced by a DFS. We can determine from the figure that the DFS would have to have visited the right neighbor of the root before the left. Example (3) could have been produced by a DFS (notice it is a forest, not a single tree). In what order did it visit the vertices? First d, then e, f, c, b, a.

- **Strongly connected components, example proof:** One of the key properties used in the strongly connected components algorithm states that "if C_1 and C_2 are two strongly components, and there is an edge from a vertex of C_1 to a vertex of C_2, then the *first* vertex visited in C_1 has a larger f-value than any vertex in C_2." How do we prove a statement like this? Here, it works best to consider the different possible ways a DFS might visit the vertices. We can break them into two cases: either (1) a vertex of C_1 is the first vertex among C_1 and C_2 to be visited, or (2) a vertex of C_2 is the first vertex among C_1 and C_2 to be visited. These two cases cover all the possibilities.

In the first case, call v the first vertex visited in C_1. By the "white-path theorem," we must eventually traverse an edge leading into C_2 before finishing v. At this point, we visit and finish all vertices in C_2, because C_2 is strongly-connected. Sometime after this point, we backup to v and finish it. So $f(v)$ is larger than the f-value for any vertex in C_2, as required.

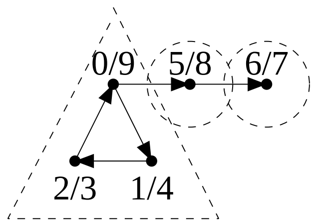
In the second case, we visit some vertex in C_2 first, and then we know that we will visit all of C_2, since it is a strongly-connected component. We also know that we cannot get to any vertex in C_1 until we visit and finish all vertices in C_2, because otherwise there would be an directed path from some vertex in C_2 to a vertex in C_1, which would make them a *single* connected component, contrary to assumption. So all the vertices in C_2 are finished before we even *visit* a vertex of C_1, so if v is the first vertex visited in c_1, we have $f(v) > d(v)$ and $d(v)$ greater than the f-value for any vertex in C_2, as required.

- **Strongly connected components, example:** We worked through the following example of the strongly connected components algorithm. The first DFS occurs in G^T, pictured on the right. The discovery and finishing times are recorded. In the process, we record a list of vertices in descending order of finishing times; in this example the list is a, c, d, f, e, g, b.



Now the second DFS is run in G, using the list of decreasing finishing times as a guide. We start at a, and discover a, b and c (the first strongly connected component). At this point the DFS gets "stuck," as intended, and restarts at an unvisited vertex. The unvisited vertex in the list with the next highest finishing time is d. Starting from d, we discover d, e and f (the second strongly connected component). Again, we get stuck, and restart at the last unvisited vertex g, which is a strongly connected component by itself.

- **Strongly connected components, practice:** Why do we go to the trouble of reversing all the graph edges for the first DFS and then computing the second DFS in descending order of finishing values? Couldn't we just do the first DFS on the original graph and compute the second DFS in order of *increasing* finishing values? Here is a counterexample:



The discovery and finishing times are recorded for each vertex, for an initial DFS starting at the leftmost vertex. The strongly connected components are marked with dashed lines. Notice that the *earliest* finishing value occurs in the strongly connected component that is a *source* in the strongly connected component DAG. If we ran the second DFS according to increasing finishing times, we would start in this component and group *all* of the graph into a single strongly connected component, which is incorrect.

Is there a counterexample to the conjecture that we can find strongly connected components by computing the second DFS in order of decreasing *discovery* times? How about increasing discovery times?

Chris Umans *February 9, 2000*
Office Hours: 3:00–5:00 Tuesdays and Thursdays in Soda 583
`umans@cs.berkeley.edu`

Sections: 10:00–11:00 in Cory 289
11:00–12:00 in Etcheverry 3109

- **Prims and Dijkstra's algorithms, review**: It is useful to separate the *strategy* behind the two algorithms from the *implementation details*. For Prim's algorithm, the idea is simple: at each step, we add the lightest edge that grows the tree. For Dijkstra's algorithm, the idea is similar: at each step, we add an edge that grows the shortest path tree; we pick an edge from the existing tree to a vertex with the shortest "shortest path estimate."

  The implementation uses *priority queues* to ensure that at each step, we can *efficiently* identify a suitable edge to add. Recall the three operations we need our priority queues to support:

  1. Insert: add a vertex to the queue.
  2. DeleteMin: remove the vertex with the smallest key.
  3. DecreaseKey: decrease the key of some vertex in the queue.

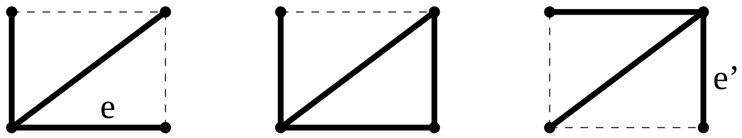
  A simple implementation of priority queues uses min-heaps. Recall that a min-heap is a complete binary tree with any unfilled positions grouped on the right of the row of leaves. This tree satisfies the *heap property*: the key of each parent has a key that is at most the key value of its children. To Insert an element, we place it in the next unfilled position at the bottom of the tree, and "percolate-up," swapping it with its parent until the heap property is satisfied. To perform a DeleteMin, we remove the root of the tree, replace it with the rightmost leaf element, and then "sift down," exchanging this element with its parent until the heap property is satisfied. To perform a DecreaseKey, we change the key value and "percolate up" until the heap property is satisfied.

  The main fact used in analyzing the running times of these operations is that the depth of the heap is $O(\log n)$, where n is the number of nodes in the heap. Since all of the operations involve moving an element (at worst) from the top to the bottom of the tree, or vice versa, all three of the operation run in time $O(\log n)$.

- **Exchange Property, proof**: The Exchange Property is a useful tool in proving things about Minimum Cost Spanning Trees (MCSTs). It states:

Lemma 2 (Exchange Property) *Given spanning trees T and T', for any edge $e' \in T' - T$, there exists an edge $e \in T - T'$, with the property that $T - \{e\} \cup \{e'\}$ is a spanning tree.*

Notice that we haven't mentioned anything about edge weights yet. Here is a simple example:



(a) (b) (c)

Figure (a) pictures the graph with tree T in bold; figure (c) pictures the graph with tree T' in bold. In this example, we have chosen the labeled edge $e' \in T' - T$, and we want to perform an exhange using this edge. We can think of first adding edge e' to T (pictured in figure (b)), and then removing the edge $e \in T - T'$ whose existence is guaranteed by the lemma.

How do we prove this lemma? It might be useful to use the example as a guide. We first notice that in the general case, adding edge e' to T creates a unique cycle, because T is a spanning tree. Then to restore a spanning tree, we need only to remove a single edge e on that cycle. The lemma requires that this edge be in $T - T'$. What would have to happen for us *not* to be able to find an edge $e \in T - T'$ on the cycle? What could go wrong? All of the edges on the cycle are already in T, so it would have to be that all of the edges were also in T'. But that can't happen, because T' is a tree! So an edge e as required by the lemma must exist, and that proves the lemma.

- **Transforming a spanning tree into a MCST**: A very useful Theorem one can prove using the Exchange Property is the following:

  Theorem 3 *Let T be a spanning tree and let M be a MCST. Using a series of exchange moves, we can transform T into M:*

  $$T \to T_1 \to T_2 \to T_3 \to \cdots \to T_k = M.$$

  Furthermore, the weights of each successive spanning tree are non-increasing; i.e. $w(T_i) \geq w(T_{i+1})$ for all i.

  Proof: The proof is by induction on the number of edges T and M share. As a base case, if they share all $n-1$ edges, then $T = M$, and the theorem is trivially true.

Otherwise, pick e' to be the *lightest* edge that is *not in both* trees. That is, e' is the lightest edge among all those edges in $T - M$ and $M - T$. We'll perform an exchange on edge e' and see what we end up with.

Since e' comes from either M or T, there are two cases:

1. $e' \in T - M$: if we perform an exchange using the edge $e \in M - T$ guaranteed by the Exchange Property, we obtain a new spanning tree $M' = M - \{e\} \cup \{e'\}$. Like e', e is an edge that is not in both trees. Since e' was chosen to be the lightest such edge, we know that $w(e') \leq w(e)$. We conclude that $w(M') \leq w(M)$, and because M was a MCST, it must be that equality holds, and therefore M' is also a MCST. Tree M' has more edges in common with T than M did, so by induction, we can identify the required sequence of exchanges to transform T into M', and then reverse the exchange we just performed to get M.

2. $e' \in M - T$: if we perform an exchange using the edge $e \in T - M$ guaranteed by the Exchange Property, we obtain a new spanning tree $T' = T - \{e\} \cup \{e'\}$. As before, we know that $w(e') \leq w(e)$. We conclude that $w(T') \leq w(T)$, so this exchange is a perfectly good first step for our sequence. Tree T' has more edges in common with M than T did, so by induction, we can identify the required sequence of exchanges to transform T' into M, completing the sequence of exchanges required by the theorem.

■

- **Application of Theorem 3**: The question we want to answer is the following: Suppose we have a graph G in which all edge weights are distinct, and let M be a MCST of G. Is M unique?

  The answer turns out to be "yes," but it is not so straightforward to prove this from scratch. Using Theorem 3 above, it becomes pretty simple.

  Suppose, for the purpose of contradiction, that we have two distinct MCSTs M_1 and M_2. Applying the theorem, we see that we can transform M_1 into M_2 using a series of exhange moves, in which the weights of successive trees are non-increasing. But the very first exchange in this sequence, which transforms M_1 into some other spanning tree M_1', must alter the weight *somehow*, since the two edges being exchanged have distinct weights. Since M_1 is a MCST, it cannot be the case that the weight decreases. Therefore it must increase, which contradicts Theorem 3. We conclude that there cannot have been two distinct MCSTs.

Chris Umans *February 16, 2000*
Office Hours: 3:00–5:00 Tuesdays and Thursdays in Soda 583
umans@cs.berkeley.edu

Sections: 10:00–11:00 in Cory 289
11:00–12:00 in Etcheverry 3109

- **Union-Find Analysis**: We want to analyze the number of steps required to perform m operations (Unions or Finds) on n elements. We assume that we use both the Union-By-Rank and Path Compression heuristics:

  1. Union-By-Rank: each element has a RANK value, which we should think of as an *overestimate* (why?) of the height of that element's subtree. Singleton sets start out with a RANK of 0. When we Link two subtrees, we always link the one with smaller RANK to the one with larger RANK. If both subtrees have the same RANK, then it doesn't matter which we link to which; however, the RANK of the new root of the combined tree is incremented in this situation. Note that this is the *only* scenario in which an element's RANK can change.
  2. Path Compression: when we perform a Find on an element x, we follow its parent pointers up the tree to the root. Path Compression changes the parent pointers of these elements (the ones visited traversing the path from x to the root) to point *directly* to the root, to reduce the work in future Find operations.

  We begin by proving two properties of RANKs.

  Observation 1 *The number of elements of RANK k is at most $n/2^k$.*

  Proof: Let x be an element with RANK k. We argue that the number of elements in the tree rooted at x is at least 2^k. We prove this by induction on k. For the base case, if $RANK(x) = 0$, we have only a single element, so the tree rooted at x has size $1 = 2^0$. For the general case, we ask "How could x get a RANK of k?" As noted above, this can only happen if we link two trees, each with RANK $k - 1$. By the induction hypothesis the size of each of these (disjoint) trees is $\geq 2^{k-1}$. Therefore the size of the resulting tree (with RANK k) is at least $2 \times 2^{k-1} = 2^k$.

  Now, we notice that the ranks of elements *below* x in the tree have RANKS *strictly smaller* than k. Therefore, for every element x of RANK k, we must have at least 2^k *other* elements of which don't have RANK k in the tree below x. Therefore, if there are n total elements, we can only accomodate $n/2^k$ elements of RANK k, at most. ■

Observation 2 *The RANKS of the n elements range from* 0 *to* $\log n$.

Proof: Clearly, no RANK can be smaller than 0, since all elements start with RANK 0, and RANKs can only increase. Now suppose that the largest RANK r was greater than $\log n$. Then by Observation 1, the number of elements of RANK r is at most $n/2^r < 1$. Since the number of elements of RANK r is an integer, it must be 0; therefore, the largest rank is at most $\log n$. ■

Now, we group the RANKs as follows: RANK k is in group $\log^*(k)$, where $\log^*$ is the iterated log function. Notice that each group contains RANKs that go from $k+1$ to 2^k, where k is some "tower of twos" (e.g. $2^{2^{2^2}}$). As in the readings, we will sometimes refer to a group by the range of RANKs that it contains, like this: $(k, 2^k]$.

We can now make two further observations about RANKs and groups:

Observation 3 *There are at most* $\log^* n$ *different groups.*

Proof: By Observation 2, the largest RANK is at most $\log n$. Therefore, the largest group number is $\log^*(\log n)$ which equals $(\log^* n) - 1$. Since the groups are numbered starting at 0, we have at most $(\log^* n) - 1 + 1 = \log^* n$ distinct groups. ■

Observation 4 *The number of elements in group* $(k, 2^k]$ *is at most* $n/2^k$.

Proof: The elements in group $(k, 2^k]$ have RANKs between $k+1$ and 2^k, inclusive. By algebra, and Observation 1, we have:

$$\sum_{j=k+1}^{2^k} n/2^j \leq \sum_{j=k+1}^{\infty} n/2^j = n/2^k.$$

■

Now, we are ready to bound the number of steps in m operations on n elements. Since all of the work in a Union operation is done in the two Finds that it requires, we will simply analyze the amount of work done in m Finds, and we will only be off by a constant.

The work done during Find(x) is simply the number of steps we must travel to get from x to the root. We will account for these steps *summed over all m operations* using *ammortized analysis*. This is just an accounting trick that makes the analysis easier to think about: we will "charge" some steps to the Find operations themselves, and some steps we will "charge" to particular elements. In the end, the sum of all of these charged steps is just the total number of steps performed, which is what we want.

For each Find(x) operation, we will charge the steps as follows. Remember that we are traveling from x to the root, and as we go, we encounter elements with larger and larger RANKs. For those pointers that we follow that go from an element whose RANK is in group i to an element whose RANK is in group $i+1$, we charge one step to the Find operation. Since there are at most $\log^* n$ different groups (by Observation 3), the total charge to each Find operation is at most $\log^* n$.

For the other pointers that we follow while performing a Find(x) operation, we charge a step to the element from which the pointer originates. Suppose u is some vertex on the path from x to the root. Notice that every time we follow a pointer out of u, the RANK of u's *parent* changes (because of path compression), unless u's parent is already the root. This gives us a way to bound the number of times we can follow a pointer out of u before the RANK of u's parent changes to a new group. Specifically, we can follow a pointer out of u at most 2^k times while u's parent is still in group $(k, 2^k]$.

Now, let us sum over all of the elements in group $(k, 2^k]$. By Observation 4, there are at most $n/2^k$ of these elements, and by the previous paragraph, we charged each of them at most 2^k times. This leads to a total charge among all elements of $n/2^k \times 2^k = n$, while the parent RANKs remain in group $(k, 2^k]$.

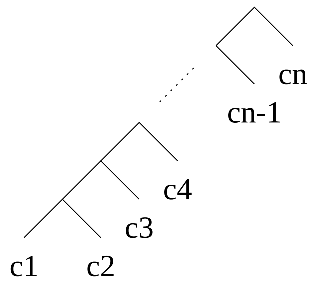
As we perform more operations, the parent RANKs move to higher and higher groups – remember that RANKs (and therefore group numbers) never decrease. So to account for the total charge, we need to sum over the at most $\log^* n$ groups, giving us a total charge among all elements during all m operations of at most $n \log^* n$.

Finally, summing the charge to the m Find operations, and the charge to the elements, we get that the total work for m operations is $O((m+n)\log^* n)$.

Chris Umans *February 23, 2000*
Office Hours: 3:00–5:00 Tuesdays and Thursdays in Soda 583
`umans@cs.berkeley.edu`

Sections: 10:00–11:00 in Cory 289
11:00–12:00 in Etcheverry 3109

- **Huffman codes for Fibonacci frequencies**: In the second part of the short exercise, we are given n characters $c_1, c_2, \dots c_n$ whose frequencies are Fibonacci numbers, i.e. $f(c_i) = F_i$, where F_i is given by the recurrence: $F_1 = 1, F_2 = 1$ and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 3$. Based on the first part of the exercise, we guess that this will result in a highly unbalanced Huffman tree, that looks like this:



This would result in a Huffman code of: 0^{n-1} for c_1, $0^{n-2}1$ for c_2, and $0^{n-i}1$ for c_i, $i \geq 3$.

How do we *prove* that this tree gets built? Recalling the example on a small number of characters in the first part of the exercise. We should show that at each step of the algorithm, the two smallest frequencies considered by the algorithm are the frequency at the root of the tree built so far, and the next smallest frequency among the individual characters not yet in the tree. More formally, we need to show that for all $i \geq 1$, at step i of the algorithm, there is a single tree consisting of all of the characters $c_1, c_2, \dots c_i$, and that the frequency at the root of this tree, $\sum_{j=1}^{i} F_j$, is strictly less than the F_{i+2}. If this is true, then the next two trees linked by the algorithm are the tree so far, and the single character c_{i+1}, with frequency F_{i+1}.

So, what we need to prove boils down to this: $\sum_{j=1}^{i} F_j < F_{i+2}$, for all $i \geq 1$. This cries out for an induction proof. The base case, when $i = 1$, is simple:

$$\sum_{j=1}^{i} F_j = \sum_{j=1}^{1} F_j = F_1 = 1 < 2 = F_3 = F_{i+2}.$$

For the induction step, consider the definition of F_{i+2}: it is equal to $F_{i+1} + F_i$. We can replace one of these with a sum and make it smaller, using the induction hypothesis. After trying both, it seems that it is easiest to replace F_{i+1}, giving us:

$$F_{i+2} = \underbrace{F_{i+1} + F_i > \left(\sum_{j=1}^{i-1} F_j\right) + F_i}_{\text{by induction}} = \sum_{j=1}^{i} F_j.$$

That completes the proof.

- **Greedy Choice Property**: The book "defines" something it calls the Greedy Choice Property, which is supposed to give you a way to determine if a greedy strategy will work on a given problem. This is sort of a vague concept, that translates roughly into: "The Greedy Choice Property holds for a particular greedy algorithm if the greedy choice made at each step of that algorithm can't hurt you." We looked at three examples of greedy algorithms, and the statements of the Greedy Choice Property for each.
  - **Prim's MST algorithm**: Here, the Greedy Choice Property states that if tree X is contained in some MST of G, and e is the lightest edge in G that extends X, then $X \cup \{e\}$ is also contained in some MST of G.

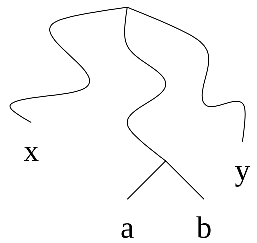
    When we apply the Greedy Choice Property to prove correctness of Prim's algorithm, X is the tree grown so far, and we apply the statement above to show that after each step of the algorithm the tree so far is contained in some MST. Therefore, after the last step, we must have a MST.
  - **Kruskal's MST algorithm**: The Greedy Choice Property here is very similar to the one for Prim's algorithm. It states that if *forest* X is contained in some MST of G, and e is the lightest edge in G *whose addition to* X *does not create a cycle*, then $X \cup \{e\}$ is contained in some MST.

    We apply this statement to prove the correctness of Kruskal's in exactly the same way we used the Greedy Choice Property to prove correctness of Prim's algorithm.
  - **Huffman's algorithm**: The Greedy Choice Property here states that if x and y are the characters with the two lowest frequencies, then there exists some optimal Huffman tree in which x and y are siblings.

    We use this to prove correctness as follows: the Greedy Choice Property tells us that there *exists* an optimal Huffman tree T in which x and y are siblings. A separate argument says that if T is an optimal tree in which x and y are siblings, then combining x and y into a single meta-character z whose frequency is $f(x) + f(y)$ gives an optimal tree for this new (smaller) character set (CLR Lemma 17.3). Therefore, at each step in the algorithm, the tree produced "so far" is contained in some optimal tree; therefore after the last step, we must have an optimal tree.

- **Proof of Greedy Choice Property for Huffman's algorithm**: It is instructive to recall how we proved the Greedy Choice property for, say, Prim's algorithm. There we wanted to show that extending X by e was "safe." We argued that if T is a MST containing X, then either (1) T also contains e, and we are done, or (2) T does not contain e. In case (2) we showed how to swap e with some other edge e' already in T to form a new tree T'; we also showed that the swap can only have *decreased* the cost, so T' must also be a MST, and it contains X and e, as required.

  For the Greedy Choice Property for Huffman's algorithm, we pursue a similar strategy. We consider an optimal tree T. If T has x and y as siblings, we are done. Otherwise, tree T looks like this:

  Here, a and b are the deepest siblings in T. We will form a new tree T'' by exchanging x with a and y with b, and argue that $B(T'') \leq B(T)$, so T'' is optimal if T was. Recall that the cost of a tree is defined by: $B(T) = \sum_{c \in C} f(c) d_T(c)$, where C is the set of characters, $f(c)$ is the frequency of character c, and $d_T(c)$ is the depth of c in tree T.

  We first consider T' obtained by swapping x and y. We will show that $B(T') \leq B(T)$ by showing that the net decrease is non-negative:

  $$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\ &= f(x) d_T(x) + f(a) d_T(a) - f(x) d_T(a) - f(a) d_T(x) \\ &= [f(a) - f(x)][d_T(a) - d_T(x)] \end{aligned}$$

  Since x and y were chosen to be the characters with the smallest frequencies, we know that $f(x) \leq f(a)$, and by our choice of a and b, we know that $d_T(x) \leq d_T(a)$.

Therefore both of the terms in the product on the last line above are non-negative, so $B(T) - B(T')$ is non-negative.

Finally, we transform T' into T'' by swapping y and b, and using an identical argument, we show that $B(T'') \leq B(T)$. Here we use the fact that $f(y) \leq f(b)$, and $d_T(y) \leq d_T(b)$. The result is an optimal tree T'' that includes x and y as siblings, which shows that the Greedy Choice Property holds for the Huffman algorithm.

Chris Umans *March 1, 2000*
Office Hours: 3:00–5:00 Tuesdays and Thursdays in Soda 583
`umans@cs.berkeley.edu`

Sections: 10:00–11:00 in Cory 289
11:00–12:00 in Etcheverry 3109

- **Strassen's algorithm example**: In the short exercises you are asked to multiply two 2×2 matrices using Strassen's algorithm. It's a good idea to first compute the result using ordinary matrix multiplication to check our results:

$$\underbrace{\begin{bmatrix} 1 & 3 \\ 5 & 7 \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} 8 & 4 \\ 6 & 2 \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} 26 & 10 \\ 82 & 34 \end{bmatrix}}_{C}.$$

Now we move on to Strassen's method. We first need to compute the 7 intermediate results (using a total of 7 multiplications):

$$\begin{aligned}
P_1 &= (a_{11} - a_{22})(b_{21} + b_{22}) = (3 - 7)(6 + 2) = -32 \\
P_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) = (1 + 7)(8 + 2) = 80 \\
P_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) = (1 + 5)(8 + 4) = -48 \\
P_4 &= (a_{11} + a_{12})b_{22} = (1 + 3)2 = 8 \\
P_5 &= a_{11}(b_{12} - b_{22}) = 1(4 - 2) = 2 \\
P_6 &= a_{22}(b_{21} - b_{11}) = 7(6 - 8) = -14 \\
P_7 &= (a_{21} + a_{22})b_{11} = (5 + 7)8 = 96
\end{aligned}$$

Now, using addition and subtraction of the intermediate results, we compute the 4 entries of the result matrix C:

$$\begin{aligned}
c_{11} &= P_1 + P_2 - P_4 + P_6 = (-32) + 80 - 8 + (-14) = 26 \\
c_{12} &= P_4 + P_5 = 8 + 2 = 10 \\
c_{21} &= P_6 + P_7 = (-14) + 96 = 82 \\
c_{22} &= P_2 - P_3 + P_5 - P_7 = 80 - (-48) + 2 - 96 = 34
\end{aligned}$$

As expected, these numbers agree with our computation of $C = AB$ above.

- **Recursive algorithms for matrix multiplication – why they work**: Strassen's algorithm is one example of a *recursive* algorithms for matrix multiplication. There are two fundamental (and not necessarily obvious) facts about matrices that justify the correctness of these recursive algorithms.

Fact 1 *We can subdivide $n \times n$ matrices M_1 and M_2 and compute their product piece by piece.* This is called *blocking.* Strassen's algorithm divides matrices into 4 $n/2 \times n/2$ matrices, which is the case we will verify here. But one can just as easily block matrices in other ways (e.g. into 9 $n/3 \times n/3$ submatrices). Let us consider the blocking used by Strassen's algorithm:

$$\underbrace{\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]}_{M_1} \times \underbrace{\left[\begin{array}{c|c} E & G \\ \hline F & H \end{array}\right]}_{M_2} = \underbrace{\left[\begin{array}{c|c} AE+BF & AG+DH \\ \hline CE+DF & CG+DH \end{array}\right]}_{M_1 M_2}.$$

Clearly, if A, B, C, D, E, F, G and H were just *numbers*, the multiplication above is correct – we would just be multiplying 2×2 matrices, the usual way. The amazing fact is that this also is correct if $A, B, C, \ldots H$ are entire *matrices.* To see this, consider the ij-th entry in the result matrix $M_1 M_2$. Say it lies in the upper-left quadrant (an identical argument can be made for the case in which it lies in any of the other quadrants). We know that it should be the dot-product of the i-th row of M_1 and the j-th column of M_2. The way we are computing it here, though, is by adding the ij-th entry of the *matrix product* AE to the ij-th entry of the *matrix product* BF. But what are these values? The ij-th entry of AE is simply the dot product of the *first half* of M_1's row i with the *first half* of M_2's column j. The ij-th entry of BF is simply the dot product of the *second half* of M_1's row i with the *second half* of M_2's column j. When we add them together, we get the dot product of M_1's row i with M_2's column j, which is exactly the correct value for the ij-th entry of $M_1 M_2$!

Fact 2 *Arithmetic works with matrices in place of numbers.* For example, if I write: $(a+b)c - bc$, I know by algebra that I can rewrite this as $(a+b)c - bc = (ac+bc) - bc = ac + (bc - bc) = ac$. I am assuming a, b and c are, say, real numbers. It turns out that one can do the same algebraic manipulations when a, b and c are *matrices.* We know how to multiply and add matrices, and one can verify that the various rules used in grade-school algebra (multiplication distributes over addition, associativity, etc...) hold for matrices as well. In the terminology of abstract algebra, we say that matrices form a *ring.*

Fact 1 allows us to apply a method for multiplying 2×2 matrices *recursively.* Fact 2 is the reason we can verify algebraically that Strassen's fancy method of getting the four entries in the result matrix, and then be sure that the algebra still works out when the "entries" are entire matrices instead of just single numbers.

- **More on recursive matrix multiplication**: In presenting Strassen's algorithm as an improvement over the "naive" recursive matrix multiplication, CLR and the readings are excited that Strassen's method is able to reduce 8 multiplications to 7. But Strassen's method also uses 18 *additions*, as opposed to only 4 in the naive algorithm. Why do we care so much about multiplications, and not additions?

The answer is that we apply Strassen's 2×2 matrix multiplication method *recursively*. Each multiplication is actually a *matrix multiplication*, which is very costly, while each addition is just a matrix addition, which is comparatively much faster. It is instructive to look at the running time recurrences in each case:

$$\begin{aligned}
\text{Naive method (8 mults, 4 additions):}\quad T(n) &= 8T(n/2) + \overbrace{4(n/2)^2}^{\Theta(n^2)} \\
\text{Strassen's method (7 mults, 18 additions):}\quad T(n) &= 7T(n/2) + \underbrace{18(n/2)^2}_{\Theta(n^2)}
\end{aligned}$$

Using the Master Theorem for recurrences, we see that both recurrences fall into the case in which the number of recursive calls dominates, and the solutions are $T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$ and $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.81\cdots})$, respectively. We now can see why multiplications matter so much more than additions. The number of multiplications translates directly to the number of *recursive calls* (which dictates the running time for these recurrences). The number of additions, on the other hand simply contributes to the "$f(n)$" term in the recurrences. No matter what constant this number is, the $f(n)$ term will always be $\Theta(n^2)$, so increasing the number of additions does not hurt the asymptotic running time at all.

- **Other Strassen-like results**: Victor Pan has a method for multiplying 68×68 matrices using only $132,464$ multiplications (the naive method uses $314,432$). He also has a method for multiplying 70×70 matrices using only $143,640$ multiplications, and 72×72 matrices using only $155,424$ multiplications. And you probably thought Strassen must have spent too much time with matrices!

  There is an important point here, though. Pan's method for multiplying, say, 72×72 matrices can be applied *recursively*, in the same way Strassen's method can. We simply need to block the matrices into $n/72 \times n/72$ submatrices at each recursive call! This gives a general algorithm for multiplying matrices, with a running time recurrence of:

  $$T(n) = 155,424T(n/72) + \alpha(n/72)^2.$$

  Here α is the number of additions Pan's method requires. Notice that we don't even need to know this number, since we can see that the $\alpha(n/72)^2$ term is just $\Theta(n^2)$, and that is all the information we need to solve the recurrence. The solution of this recurrence is, by the Master Theorem, $T(n) = \Theta(n^{\log_{72} 155{,}424}) = \Theta(n^{2.795\cdots})$. This is indeed a small improvement over Strassen's algorithm. But notice how much harder Pan needed to work on larger matrices: Strassen only needed to save a single multiplication, while Pan needed to save over *half* of the multiplications (some 217,824 multiplication saved)!

Question 1 on Homework 7 asks you to figure out how many multiplications you would need to save on 3×3 matrices in order to improve Strassen's result.

Chris Umans *March 8, 2000*
Office Hours: 3:00–5:00 Tuesdays and Thursdays in Soda 583
umans@cs.berkeley.edu

Sections: 10:00–11:00 in Cory 289
11:00–12:00 in Etcheverry 3109

- **Discrete Fourier Transform example**: In the short exercises you are asked to compute the Discrete Fourier Transform (DFT) of the vector $(0, 1, 2, 3)$. Some people elected to do this by tracing through the FFT algorithm. Here we will simply do the appropriate polynomial evaluations, which seems a less error-prone method.

  Remember that the input vector $a = (a_0, a_2, \ldots a_{n-1})$ gives the coefficients of a polynomial $A(x)$. Here we have:

  $$A(x) = 0 + 1 \cdot x + 2 \cdot x^2 + 3 \cdot x^3.$$

  The DFT requires us to evaluate the polynomial $A(x)$ at the n complex n-th roots of unity $\omega_n^0, \omega_n^1, \omega_n^2, \ldots \omega_n^{n-1}$. Here we notice[1] that the complex 4-th roots of unity are just $1, i, -1, -i$. We compute the DFT vector $y = (y_0, y_1, y_2, y_3)$ as follows:

  $$\begin{aligned} y_0 &= A(\omega_4^0) = A(1) = 0 + 1 \cdot (1) + 2 \cdot (1)^2 + 3 \cdot (1)^3 = 6 \\ y_1 &= A(\omega_4^1) = A(i) = 0 + 1 \cdot (i) + 2 \cdot (i)^2 + 3 \cdot (i)^3 = -2 - 2i \\ y_2 &= A(\omega_4^2) = A(-1) = 0 + 1 \cdot (-1) + 2 \cdot (-1)^2 + 3 \cdot (-1)^3 = -2 \\ y_3 &= A(\omega_4^3) = A(-i) = 0 + 1 \cdot (-i) + 2 \cdot (-i)^2 + 3 \cdot (-i)^3 = -2 + 2i. \end{aligned}$$

  Therefore the DFT of $(0, 1, 2, 3)$ is $(6, -2 - 2i, -2, -2 + 2i)$.

- **The Fast Fourier Transform (FFT) algorithm**: This is an incredibly important and beautiful algorithm, that computes the DFT in time $O(n \log n)$, instead of the $O(n^2)$ time we get using the "obvious" algorithm of just performing the n seperate polynomial evaluations.

  Our goal is to evaluate a polynomial $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$ at the complex n-th roots of unity. For the application we have in mind (multiplying polynomials), *any* n distinct points will do, but as we will see, these particular n points are critical for the divide and conquer approach we will take. The complex

[1] We can see this by plugging in 4 for n in the following two equations: (1) $\omega_n = e^{2\pi i/n}$, and (2) $e^{iu} = \cos(u) + i \sin(u)$.

n-th roots of unity are: $1 = \omega_n^0, \omega_n^1, \omega_n^2, \ldots \omega_n^{n-1}$, where ω_n is defined to be $e^{2\pi i/n}$. It is important to remember that these are simply complex numbers, nothing more.

The FFT is a divide and conquer algorithm. As with any divide and conquer algorithm, we first need to specify how to split the problem into subproblems. Our input is an $(n-1)$-degree polynomial $A(x)$, which we will split into two $(n/2-1)$-degree polynomials as follows:

$$A(x) = A^{\text{even}}(x^2) + xA^{\text{odd}}(x^2), \tag{1}$$

where A^{even} and A^{odd} simply use the even and odd coefficients of A, respectively (we assume n is even):

$$\begin{aligned} A^{\text{even}}(z) &= a_0 + a_2 z + a_4 z^2 + a_6 z^3 + \cdots + a_{n-2} z^{n/2-1} \\ A^{\text{odd}}(z) &= a_1 + a_3 z + a_5 z^2 + a_7 z^3 + \cdots + a_{n-1} z^{n/2-1}. \end{aligned}$$

We write A^{even} and A^{odd} as polynomials in z to emphasize that while they are *derived* from $A(x)$, they are not the same as $A(x)$, nor are they formed from a subset of the terms of $A(x)$. It is easy to verify, however, that Equation 1 above is correct; i.e. in order to evaluate A at a point x, we need only to evaluate A^{even} and A^{odd} at the point x^2, and then combine the results with one further multiplication and addition.

This leads to the following strategy for achieving our goal of computing the n values:

$$A(\omega_n^0), A(\omega_n^1), A_(\omega_n^2), \ldots A(\omega_n^{n-1}).$$

Using our way of splitting up A, we should compute:

$$A^{\text{even}}((\omega_n^0)^2), A^{\text{even}}((\omega_n^1)^2), A^{\text{even}}((\omega_n^2)^2), \ldots A^{\text{even}}((\omega_n^{n-1})^2),$$

and

$$A^{\text{odd}}((\omega_n^0)^2), A^{\text{odd}}((\omega_n^1)^2), A^{\text{odd}}((\omega_n^2)^2), \ldots A^{\text{odd}}((\omega_n^{n-1})^2),$$

and finally combine each of the n pairs: $A(\omega_n^k) = A^{\text{even}}((\omega_n^k)^2) + \omega_n^k A^{\text{odd}}((\omega_n^k)^2)$, for $k = 0, 1, 2, \ldots n-1$.

Now, for the first time, we will use a property of the special set of n numbers at which we chose to evaluate A. It would seem that our divide and conquer approach has us evaluating A at n points by evaluating A^{even} and A^{odd} at n points, which isn't an improvement. However, by the magic of complex roots of unity, it turns out that we are actually evaluating A^{even} and A^{odd} at *only $n/2$ distinct points*, and, further, these $n/2$ distinct points are exactly the complex $n/2$-th roots of unity! This is called the Halving Lemma in CLR, and we restate it here:

Lemma 4 *Let $\omega_n^0, \omega_n^1, \ldots \omega_n^{n-1}$ be the complex n-th roots of unity, and let $\omega_{n/2}^0, \omega_{n/2}^1, \ldots \omega_{n/2}^{n/2-1}$ be the complex $n/2$-th roots of unity. We have:*

$$\begin{array}{lllll}
(\omega_n^0)^2 & = & (\omega_n^{n/2})^2 & = & \omega_{n/2}^0 \\
(\omega_n^1)^2 & = & (\omega_n^{n/2+1})^2 & = & \omega_{n/2}^1 \\
(\omega_n^2)^2 & = & (\omega_n^{n/2+2})^2 & = & \omega_{n/2}^2 \\
\vdots & & \vdots & & \vdots \\
(\omega_n^{n/2-1})^2 & = & (\omega_n^{n-1})^2 & = & \omega_{n/2}^{n/2-1}.
\end{array}$$

We now have a bona fide divide and conquer algorithm. It goes as follows: to evaluate an $(n-1)$-degree polynomial $A(x)$ at the complex n-th roots of unity, *recursively* evaluate A^{even} and A^{odd}, two $(n/2-1)$-degree polynomials, at the complex $n/2$-th roots of unity; then combine the results according to Equation 1 using n multiplications and additions. The running time $T(n)$ of this algorithm is expressed by the following (familiar) recurrence:

$$T(n) = 2T(n/2) + \Theta(n),$$

whose solution, by the Master Theorem, is $T(n) = \Theta(n \log n)$.

- **Application: polynomial multiplication**: Our motivation for trying to get a fast algorithm for the DFT was polynomial multiplication. Here we are given two polynomials $A(x)$ and $B(x)$, each of degree at most n and we want to compute $C(x) = A(x)B(x)$. We noticed that just as we can uniquely express a degree n polynomial with its $n+1$ coefficients, we can also uniquely express a degree n polynomial with its *evaluation* at $n+1$ distinct points. Furthermore, given an evaluation of $A(x)$ and $B(x)$ at $2n+1$ distinct points $x_1, x_2, \ldots x_{2n+1}$, we can obtain an evaluation of $C(x) = A(x)B(x)$ at the same set of $2n+1$ distinct points by pointwise multiplication. That is, if $A(x_i) = y_i$ and $B(x_i) = z_i$, then $C(x_i) = A(x_i)B(x_i) = y_i z_i$. Finally, since $C(x)$ can have degree at most $2n$, these $2n+1$ points *uniquely* determine $C(x)$, and we can recover the coefficients by interpolation.

  Our algorithm for polynomial evaluation is thus: (1) *evaluate* $A(x)$ and $B(x)$ at $2n+1$ distinct points, (2) multiply them pairwise to obtain the value of $C(x) = A(x)B(x)$ at $2n+1$ distinct points, and (3) *interpolate* to find the coefficients of $C(x)$. Step (2) takes $\Theta(n)$ operations. The FFT algorithm allows us to do step (1) and step (3) in time $\Theta(n \log n)$ (using the complex $(2n+1)$-th roots of unity as our distinct points), for an overall running time of $\Theta(n \log n)$.

- **The Inverse DFT**: For the application above, we claimed that it was possible to use the FFT algorithm to *interpolate*, or in this case, to compute the *Inverse DFT*.

Remember that the DFT transforms a vector $a = (a_0, a_1, \ldots a_{n-1})$ of coefficients of a degree $n-1$ polynomial $A(x)$ into a vector $y = (y_0, y_1, \ldots y_{n-1})$ of evaluations of $A(x)$ at the complex n-th roots of unity. The Inverse DFT computes the vector a of coefficients from the vector y of evaluations. In the polynomial-multiplication application above, we use the Inverse DFT to recover the coefficients of $C(x)$ after we obtain a vector of evaluations of $C(x)$ at the complex $(2n+1)$-th roots of unity.

In computing the DFT, we computed from the n entries of the a vector, the n entries of the y vector, whose formulas were:

$$y_k = \sum_{j=0}^{n-1} a_j (\omega_n^k)^j.$$

It turns out that we can express the a's in terms of the y's by the following formulas (proved in CLR):

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j (\omega_n^{-k})^j.$$

These are remarkably similar! It should not be surprising, then, that we can compute the Inverse DFT using the FFT algorithm in which we replace ω_n by its inverse ω_n^{-1}, and divide all of the entries in the result vector by n (although we certainly haven't *proved* that this works).

Chris Umans *March 15, 2000*
Office Hours: 3:00–5:00 Tuesdays and Thursdays in Soda 583
`umans@cs.berkeley.edu`

Sections: 10:00–11:00 in Cory 289
11:00–12:00 in Etcheverry 3109

- **Computing binomial coefficients**: In the short exercises you are asked to compute using three different methods the binomial coefficient $C(n, k)$ defined as follows:

$$\begin{aligned} C(n,k) &= C(n-1,k-1) + C(n-1,k), n > 0, k > 0 \\ C(n,0) &= 1, n \geq 0 \\ C(0,k) &= 0, k > 0. \end{aligned}$$

The recursive method follows easily from the definitions:

```
recursive_C(n, k) {
   if(k == 0) return 1;
   if(n == 0) return 0;
   return(recursive_C(n-1, k-1) + recursive_C(n-1, k));
}
```

One can prove by a straightforward induction that the running time of recursive_$C(n, k)$ is at least $C(n, k)$ which is at least exponential in the worst case. The space required is $\Omega(n)$, the maximum depth of the recursion tree.

The dynamic programming method fills in a $(n+1) \times (k+1)$ table, in which entry (i, j) is intended to be $C(i, j)$. The only thing we need to consider before coding this is what order to fill in the table. We notice that entry (i, j) depends on entries $(i-1, j-1)$ and $(i-1, j)$. If we visualize the table with entry $(0, 0)$ in the upper left corner, this means that each entry depends on the entry diagonally above and to the left of it, and the entry directly above it. If we fill in the table from top to bottom, and left to right within each row, we will be sure that when we compute entry (i, j), the table entries consulted in computing entry (i, j) are already filled in. The code follows:

```
dynamic_prog_C(n, k) {
   for(i = 0; i <= n; i++) C[i, 0] = 1;
   for(j = 1; j <= k; j++) C[0, j] = 0;
   for(i = 1; i <= n; i++)
```

```
      for(j = 1; j <= k; j++)
         C[i, j] = C[i-1, j-1] + C[i-1, j];
}
```

As with all dynamic programming algorithms, the running time is the number of entries in the table, times the amount of work to fill in each entry. In this case, we have $\Theta(nk)$ table entries, and we fill in each with $O(1)$ work, for a total running time of $\Theta(nk)$. The space required is the size of the table, or $\Theta(nk)$.

Finally, using the formula $C(n,k) = \frac{n!}{(n-k)!k!}$, we can write the following code:

```
formula_C(n, k) {
   if(k == 0) return 1;
   if(n == 0) return 0;
   x = 1;
   for(i = 1; i <= n; i++) {
      x = x * i;
      if(i == n-k) n_minus_k_fact = x;
      if(i == k) k_fact = x;
   }
   return(x/(n_minus_k_fact * k_fact));
}
```

Counting multiplications as constant time and constant space operations (which is slightly unrealistic here, because the numbers we are computing are so large), the running time is $O(n)$, and the space required is a constant. A better running time of $O(k)$ can be obtained using the observation that $n!/(n-k)! = n(n-1)(n-2)\cdots(n-k+1)$ and $k!$ can both be computed with only k multiplications.

- **Framework for dynamic programming**: In devising a dynamic programming algorithm, there are 4 steps/questions to be answered:

   1. What are the subproblems?
   2. How do we express the solution to each subproblem in terms of solutions to *smaller* subproblems?
   3. What is the correct order to fill in the dynamic programming table?
   4. What is the running time and space requirement of the algorithm?

Step 2 and sometimes step 1 are the difficult parts; once these are complete, it should be fairly easy to complete step 3, and write the actual (pseudo-) code.

A few notes on steps 1 and 2: by a *subproblem*, we mean an *instance of the original problem* with a different (smaller) input. In determining the subproblems, sometimes it is necessary to first rephrase the original problem. In figuring out the recursive expression in step 2, we use the so-called *optimal substructure* of the problem – we often consider a range of initial choices for constructing a solution, and then each choice leads to some number of subproblems. The next few examples should make these observations more concrete.

- **Chain Matrix Multiplication**: We want to multiply a chain of rectangular matrices $A_1, A_2, \ldots A_n$ with dimensions $m_0, m_1, m_2, \ldots m_n$ (matrix A_i has dimensions $m_{i-1} \times m_i$). Since the number of operations required varies with the order we choose to multiply the matrices, we want to determine the optimal way of parenthesizing the matrices. Let's see how the solution we saw in lecture fits into the framework above:

  1. The problem asks: *What is the optimal way to multiply $A_1 \times A_2 \times \cdots \times A_n$?* The subproblems ask: *What is the optimal way to multiply $A_i \times A_{i+1} \times \cdots \times A_j$?* Notice that subproblems are all instances of the same problem, with different inputs, namely: all sub-chains of matrices $A_i, A_i + 1, \ldots A_j$. There are $O(n^2)$ such subproblems, and each is defined by a pair of endpoints (i, j). Notice that the problem on the original input is just subproblem $(1, n)$.

  2. First, we define some notation: let $M(i, j)$ be the number of operations for multiplying $A_i, \ldots A_j$ in an optimal order. Now, we want to write a *recursive* expression for $M(i, j)$, so we "think recursively": the optimal grouping first breaks the matrices at some k between i and j, multiplies the left half in an optimal order, multiplies the right half in an optimal order, and then does one final multiplication of the left result matrix and the right result matrix. In our notation:

  $$\begin{aligned} M(i,j) &= \min_{i \le k < j} [M(i,k) + M(k+1,j) + \underbrace{m_{i-1} m_k m_j}_{\text{cost of final step}}] \\ M(i,i) &= 0 \quad \text{for all } i \end{aligned}$$

  3. The dynamic programming table for this example has size $O(n^2)$, and we want to place the value of $M(i, j)$ in entry (i, j), for all $1 \le i \le j \le n$. We always fill in the table from the smallest subproblem up to the largest subproblem. Here the "size" of subproblem (i, j) is simply the number of matrices in its input, or $j - i + 1$. We can verify that in our recursive expression, $M(i, j)$ depends only on

solutions to *smaller* subproblems. So, we fill in the table in the following order:

fill in $M(i,i) = 0$	for $i = 1, 2, \ldots n$	(size 1)
fill in $M(i, i+1)$	for $i = 1, 2, \ldots n-1$	(size 2)
fill in $M(i, i+2)$	for $i = 1, 2, \ldots n-2$	(size 3)
$\vdots$	$\vdots$	$\vdots$
fill in $M(i, i+(n-1))$	for $i = 1$	(size n).

4. As noted, the size of the table in this example is $O(n^2)$, and the work required to fill in each entry is $O(n)$, since we are taking the minimum over $O(n)$ values. Therefore the running time is $O(n^3)$. The space required is the size of the table $O(n^2)$.

- **Longest Common Subsequence (LCS)**: In this problem we are given two character strings $x = x_1 x_2 \ldots x_m$ and $y = y_1 y_2 \ldots y_n$, and we wish to find the *longest common subsequence* of x and y. A subsequence is just a portion of a string obtained by omitting characters. We want to find a *longest* subsequence that occurs in both x and y. For example, if $x = nonsense$ and $y = longest$, the longest common subsequence of x and y is $ones$. Let's devise a dynamic programming algorithm using the framework above:

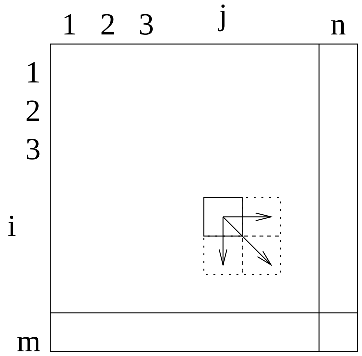
  1. The problem asks: *What is the longest common subsequence of $x_1 x_2 \ldots x_m$ and $y_1 y_2 \ldots y_n$?* The subproblems ask: *What is the longest common subsequence of $x_i x_{i+1} \ldots x_m$ and $y_j y_{j+1} \ldots y_n$?* There are $O(n^2)$ subproblems, and we can describe each by a pair of indices (i,j). As always, the subproblems are all instances of the same problem with differing inputs.

  2. We introduce the notation $L(i,j)$ to represent the *length* of the longest common subsequence in subproblem (i,j). We need to write a recursive definition for $L(i,j)$, expressing it in terms of smaller subproblems. We might start by considering the "first" choice in finding a LCS of two strings: if the first characters match, then the LCS is just the first character, plus the LCS of the remainder of the two strings. Otherwise, the LCS is either the LCS of the first string with the second string *after discarding the first character*, or the LCS of the second string with the first string *after discarding the first character*. In our notation, we have:

$$\begin{aligned} L(i,j) &= \begin{cases} 1 + L(i+1, j+1) & \text{if } x_i = y_j \\ \max[L(i+1,j), L(i,j+1)] & \text{otherwise} \end{cases} \\ L(i,n) &= 1\text{if } y_n \text{ appears in string } x_i x_{i+1} \ldots x_m \text{ and } 0 \text{ otherwise} \\ L(m,j) &= 1\text{if } x_m \text{ appears in string } y_j y_{j+1} \ldots y_n \text{ and } 0 \text{ otherwise} \end{aligned}$$

  For the base case, we noticed that the length of a LCS of two strings is easy to determine if either string is just a single character – in this case, the length of a LCS is 1 if that character appears in the other string, and 0 otherwise.

3. Like the chain matrix multiplication example, the dynamic programming table for this example has size $O(n^2)$, and we want to place the value of $L(i, j)$ in entry (i, j), for all $1 \leq i \leq j \leq n$. The order I described in section for filling in the table was slightly inaccurate. Here is a good way to visualize what the correct order should be:



Entry (i, j) in the table depends on three other entries (from our recursive definition above). It depends on the entry to its right, the entry below it, and the entry diagonally to its right and below it, as pictured. Therefore, we can fill in the table in the following order: first, fill in the last column and the last row, which correspond to the base cases. Then, fill in the table from bottom to top, and right to left within each row. It is easy to see that at the time we get to filling in each entry (i, j), the other three entries upon which it depends will have already been filled in.

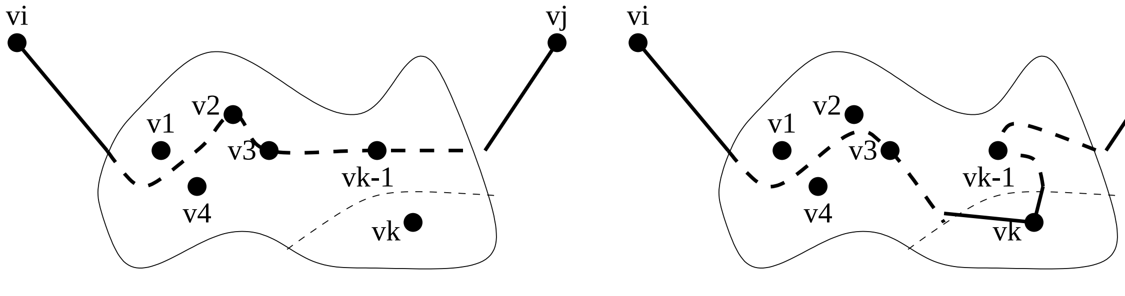
4. The size of the table is $O(n^2)$, and the time to fill in each entry is $O(1)$; therefore the running time is $O(n^2)$.

- **All-Pairs Shortest Paths**: In this problem, we are given a graph $G = (V, E)$ with edge weights $w(v_i, v_j)$ for all edges $(v_i, v_j) \in E$. (There are no negative cycles, and for convenience, we will assume $w(v_i, v_j) = \infty$ if $(v_i, v_j) \notin E$). We want to find the shortest path from v_i to v_j for *all vertex pairs* (v_i, v_j). Let's see how the dynamic programming solution to this problem fits into the framework above:

1. This is an example of a case in which we alter the problem statement in order to be able to clearly state the subproblems. Our modified problem asks: *What is the shortest path from v_i to v_j using (some of) the vertices $v_1, v_2, \ldots, v_n$ as intermediate vertices on the path?* The subproblems ask: *What is the shortest path*

from v_i *to* v_j *using (some of) the vertices* $v_1, v_2, \ldots, v_k$ *as intermediate vertices on the path?* Notice that since we will be solving *all* of the subproblems, we can extract the solution to our original problem from these solutions, since, among other things, we will have the shortest paths for *all vertex pairs* (v_i, v_j). Here there are $O(n^3)$ subproblems, and each can be described by a triple (i, j, k).

2. We introduce the notation $T(i, j, k)$ to represent the *length* of the shortest path from i to j using (some of) the intermediate vertices $v_1, v_2, \ldots v_k$. We want a recursive expression for $T(i, j, k)$ in terms of smaller subproblems. As with the LCS problem, we begin by noticing that there are two cases: either the shortest path doesn't use vertex v_k, or it does; these two possibilities are pictured below in (a) and (b), respectively:



(a) (b)

In (a), the length of the shortest path from v_i to v_j using $v_1 \ldots v_k$ as intermediate vertices is simply the length of the shortest path from v_i to v_j using vertices $v_1 \ldots v_{k-1}$ as intermediate vertices. In (b), the length is the length of the shortest path from v_i to v_k using vertices $v_1 \ldots v_{k-1}$ as intermediate vertices *plus* the length of the shortest path from v_k to v_j using vertices $v_1 \ldots v_{k-1}$ as intermediate vertices. This observation allows us to write the following recursive expression for $T(i, j, k)$:

$$\begin{aligned} T(i,j,k) &= \min[T(i,j,k-1), T(i,k,k-1) + T(k,j,k-1)] \\ T(i,j,0) &= w(v_i, v_j) \end{aligned}$$

3. Here it is difficult to visualize the dynamic programming table (since it has 3 dimensions), but it is easy to describe the order in which we must fill it in. The "size" of subproblem (i, j, k) is just k, the number of allowed intermediate vertices. It is easy to verify that in the expression for $T(i, j, k)$ above, $T(i, j, k)$ depends only on smaller subproblems. We can fill in the table in the following order:

fill in $T(i,j,0)$	for all i,j	(size 0)
fill in $T(i,j,1)$	for all i,j	(size 1)
fill in $T(i,j,2)$	for all i,j	(size 2)
$\vdots$	$\vdots$	$\vdots$
fill in $T(i,j,n)$	for all i,j	(size n)

These last entries are the answers to our original question about the shortest paths between all pairs of vertices.

4. Here, the table has size $O(n^3)$, and the work to fill in each entry is $O(1)$, for a running time of $O(n^3)$.

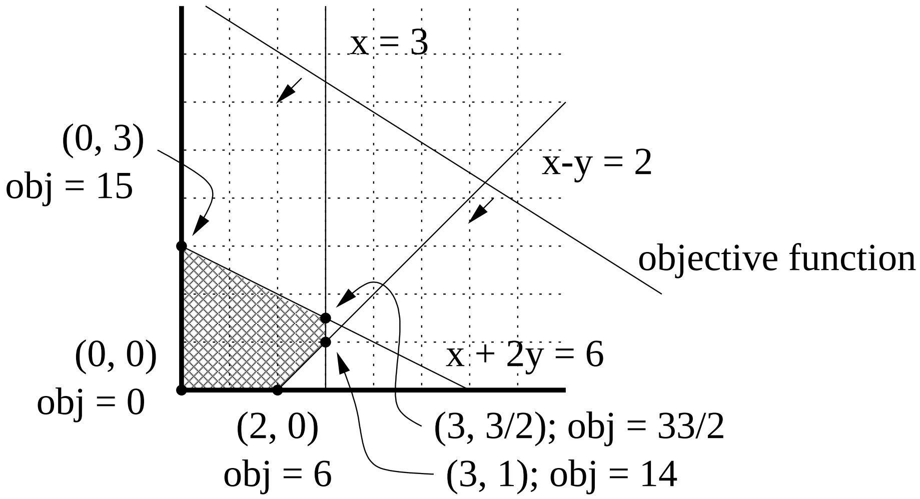
Chris Umans *March 22, 2000*
Office Hours: 3:00–5:00 Tuesdays and Thursdays in Soda 583
`umans@cs.berkeley.edu`

Sections: 10:00–11:00 in Cory 289
11:00–12:00 in Etcheverry 3109

- **Graphical LP example**: In the short exercises you are asked to solve the following Linear Program (LP) in two variables (which I have renamed x and y):

$$\begin{array}{lrcl} \text{maximize} & 3x+5y & & \\ \text{subject to} & x+2y & \leq & 6 \\ & x-y & \leq & 2 \\ & x & \leq & 3 \\ & x,y & \geq & 0 \end{array}$$

The following figure gives a graphical representation of the problem:



The line defined by each of the constraints is pictured, along with a line representing a constant value for objective function. The feasible region is shaded. We imagine "sliding" the objective function line in the direction of decreasing objective function value (in the direction of the arrows in the figure), until it first touches the feasible region. The point of intersection will be the maximum value of the objective function subject to the constraints. This will always happen at a corner (or along an entire line, in which case we can pick an endpoint of that line, which is a corner).

To determine which corner attains the maximum value of the objective function, we evaluate the objective function at each of the corners of the feasible region. In this example, there are 5 corners, each marked with a heavy dot in the figure, and each is labled with its coordinates, and the value of the objective function at that point. The optimum value of $33/2$ is attained at $(3, 3/2)$. The solution to the LP is thus: $x = 3$; $y = 3/2$.

- **Why we care**: Many real-world problems fit into the general framework "maximize or minimize some objective function, subject to some constraints." It turns out that many such formulations are very hard to solve efficiently. For example, we might want to require that the variables in an LP are integers, which is natural for many problems that seem to need a Boolean variable. We might also want an objective function that includes a *quadratic* term (one that multiplies two variables together). It turns out that both of these formulations are (probably) phenomenally hard to solve exactly. But, Linear Programming is easy, in the following two ways:

  - A Linear Program can be solved in polynomial time. The algorithms that do this are complex and often not useful in practice.
  - The *simplex* algorithm solves most Linear Programs *very efficiently* in practice, although on some degenerate cases it requires exponential running time. The simlex algorithm takes advantage of the fact that the optimum lies at a corner of the feasible region. It starts at some corner, and repeatedly moves to a better neighboring corner, until it can't improve the value of the objective function anymore. Of course, in general this algorithm is operating in n-dimensional space, moving from corner to corner of the polytope that is the feasible region. In this course, you are only expected to be able to trace the steps the simplex algorithm might take on 2 (and possibly 3) variable problems, where one can reasonably expect to be able to draw the feasible region graphically.

  Because we have an efficient way to solve LP's, and they provide a quite general framework for expressing optimization problems, it's often worth putting a fair amount of effort into trying to express an optimization problem as a LP. If such a transformation can be found, it automatically gives an efficient solution to the optimization problem.

- **Example: production scheduling**: One of the major uses of Linear Programming is in business, where a company is trying to make decisions that maximize profit, while obeying certain constraints. Here is a simple example: a business produces two products, A and B, and it must decide how many of each to produce in the next month.

Say product A sells for \$10 per unit and B sells for \$12 per unit. Then to maximize profit, the business should solve the following LP:

$$\begin{array}{lrcl} \text{maximize} & 10a + 12b & & \\ \text{subject to} & a, b & \geq & 0 \end{array}$$

Here a is the amount of A to produce and b is the amount of B to produce. What is the solution to this LP? The solution is to produce an infinite amount of A and B. In LP terminology we say that this problem is *unbounded.* Clearly this is not realistic, so we should add some constraints to model the situation more accurately. Let's say that the company can produce no more than $10,000$ units of either product. We should then add the constraints:

$$\begin{array}{rcl} a & \leq & 10000 \\ b & \leq & 10000 \end{array}$$

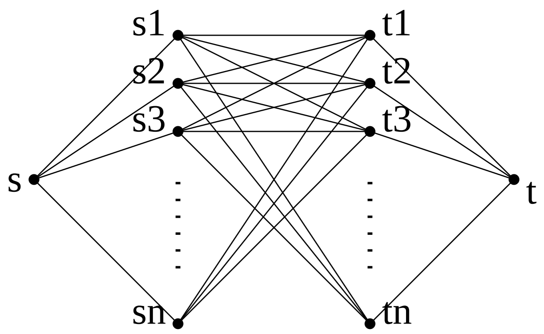
Now, the optimum is achieved by producing $10,000$ units of each. This is reasonable but not terribly interesting. Now suppose there is a single packaging machine that must package each unit of A or B produced by the company before it is shipped, and suppose that it takes 1 minute to package a unit of A and 3 minutes to package a unit of B. Furthermore, the packaging machine is run during working hours, which are 40 hrs/week, for 4 weeks. We get the following additional constraint:

$$a + 3b \leq 9600,$$

because the machine can operate for a total of $4 \times 40 \times 60 = 9600$ minutes. Now it's harder to see what the optimum is, but we could use our LP-solver to find it.

To make the problem even more realistic, we could consider planning a schedule for the next 12 months. We would then have 24 variables ($a_1, a_2, \ldots a_{12}$ and $b_1, b_2, \ldots b_{12}$) instead of 2, representing the amount of each product produced each month. Our constraints could be modified to account for anticipated different prices from month to month, different monthly costs, and the possibility of producing extra units one month and storing them (at some cost, of course) until they are needed in a future month. The readings have such an example. The point is that it is often very easy to formulate this type of decision-making as a LP, and by adding constraints, we can model quite complex scenarios.

- **Example: maximum delay in a network**: Suppose I send an email to a friend and I want to know how long I must wait before I am certain that the email has been received. The maximum delay will depend on the characteristics of the network through which the message passes. We model the network with the following simple graph:



In this model, the source computer is located at the vertex s, the destination is at vertex t, the vertices labeled $s_1, s_2, \ldots s_n$ represent the possible points of entry to the network, and the vertices labeled $t_1, t_2, \ldots t_n$ represent the possible exit points. The edges (s_i, t_j) represent possible routes through the network. Every edge e in the graph has an associated weight $w(e)$ that is an upper bound on the delay along that route. To solve my problem, I am interested in the path from s to t with the maximum weight. We will try to write this problem as a Linear Program.

This is an example of a problem where the LP variables are not at all obvious. For this problem we will have a variable associated with each vertex, which, by abuse of notation we will label with the same name as the vertex itself. So, the LP variables are $s, t, s_1, s_2, \ldots s_n, t_1, t_2, \ldots t_n$. The LP is:

$$\begin{array}{lrcll} \text{minimize} & s - t & & & \\ \text{subject to} & s - s_i & \geq & w(s, s_i) & \forall i \\ & s_i - t_j & \geq & w(s_i, t_j) & \forall i, j \\ & t_j - t & \geq & w(t_j, t) & \forall j \end{array}$$

This is fairly counter-intuitive. To understand how these constraints are useful, notice that we are trying to minimize the gap between s and t. That is, the LP is trying to push t as close to s as it possibly can. We want the gap between s and t to be the length of the longest path, so we need constraints that prevent the gap from becoming too small. Each constraint in the LP forces the gap between the variables representing endpoints of an edge to be at least as large as the length of that edge. In particular, we can prove the following claim:

Claim 1 *For all paths p from s to t, a feasible assignment to the variables in the LP above must have $s - t \geq$ length(p).*

Proof: Say the path p goes from s to s_i to t_j to t. Let's see what information we can get from the constraints we know these four variables must obey:

$$\begin{aligned} s - s_i &\geq w(s, s_i) \\ s_i - t_j &\geq w(s_i, t_j) \\ t_j - t &\geq w(t_j, t) \end{aligned}$$

If we sum these inequalities, we get $s - t \geq \text{length}(p)$ as required. ∎

Since this claim holds for all paths, it holds, in particular, for the longest path. This means that the optimum value of the LP can be no smaller than the length of the longest path. To finish proving that the optimum actually gives us the length of the longest path, we need to exhibit a feasible solution for which the objective function actually attains the value $\text{length}(p)$, where p is a longest s-t path.

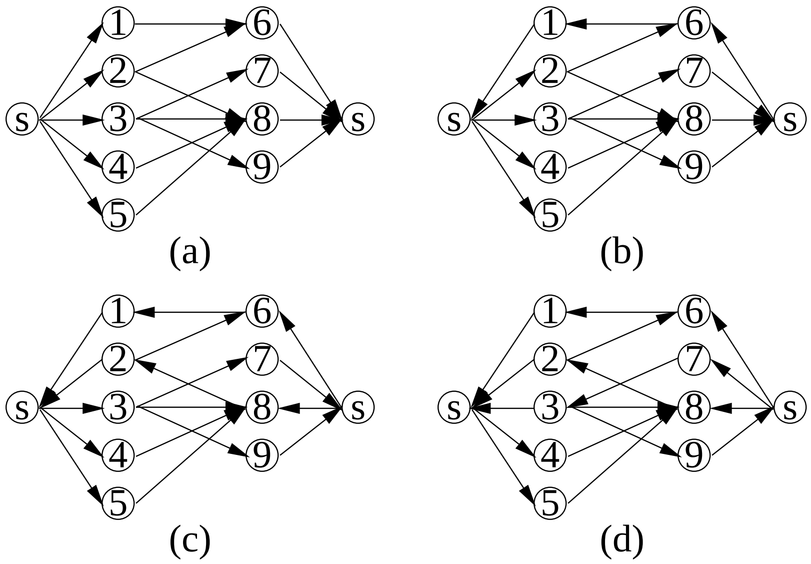
It turns out that setting $t = 0$, $t_j = w(t_j, t)$ for all j, $s_i = \max_j(w(s_i, t_j) + w(t_j, t))$ for all i, and s to the length of the longest s-t path, we get a feasible solution. The proof is not difficult, but we did not get to it in section.

This problem is meant to give you some ideas that should be useful for the last problem on the homework, which asks you to give a LP formulation of the single-pair shortest paths problem.

Chris Umans *April 5, 2000*
Office Hours: 3:00–5:00 Tuesdays and Thursdays in Soda 583
`umans@cs.berkeley.edu`

Sections: 10:00–11:00 in Cory 289
11:00–12:00 in Etcheverry 3109

- **Network flow example**: In the short exercises, you are asked to draw the residual graph after each augmenting path is added to the flow in a simple flow network. The graphs should look something like this:

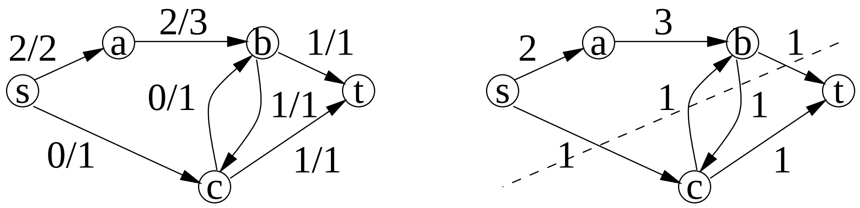


  All the edge capacities in the figures are one. Figure (a) is the original directed graph, which is the same as the residual graph with respect to the "zero" flow (no flow on any edge). The first augmenting path we find is: $s \to 1 \to 6 \to t$. We push one unit of flow along this path, and get the residual graph pictured in (b). Next we find the augmenting path $s \to 2 \to 8 \to t$, and push one unit of flow along it. This yields the residual graph in (c). Finally we find the augmenting path $s \to 3 \to 7 \to t$, and push one unit of flow along it, yielding the residual graph in (d). In this residual graph, there are no augmenting paths, so we know that the total flow of three units that we have achieved is a maximum flow.

- **Max-Flow/Min-Cut Theorem**: This is an important theorem, and working through the proof is a good way to become familiar with properties of flows and augmenting paths.

Theorem 5 (Max-Flow/Min Cut Theorem) *Given a flow network G, the value of a maximum flow in G* equals *the capacity of a minimum cut in G.*

Proof: First, a reminder of some definitions, via some examples:



(a) (b)

The *value* of a flow f, denoted $|f|$ is the amount of flow leaving the source s (equivalently, the amount of flow entering the sink t). In the example network pictured in (a), each edge is labeled with the flow along that edge and the edge capacity; the value of the flow in this example is 2.

The *capacity* of a cut (S,T), denoted $c(S,T)$ is the sum of the capacities of the edges crossing the cut in the $s \to t$ direction. (Remember that a *cut* is a partition of the vertices into two sets, S and T, such that S contains the source s and T contains the sink t). In the example network pictures in (b), the dashed line represents a cut (i.e., $S = \{s, a, b\}$ and $T = \{c, t\}$), whose capacity is 3.

We can now proceed with the proof. Our strategy is to prove the following two statements, which together imply the theorem:

1. The value of *any* flow in G is *at most* the capacity of a minimum cut in G.
2. The value of a maximum flow f in G *equals* the capacity of *some* cut in G.

Proof of (1). Let (S,T) be a minimum cut in G, and let f be any flow in G. First, we claim that:

$$|f| = \sum_{u \in S, v \in V} f(u,v).$$

Why is this true? Let's break down the sum, by considering different vertices u might be. When u is the source s, we sum all the outgoing flow from s, which we know equals $|f|$. When u is any other vertex in S, we sum the outgoing flow *minus* the incoming flow, which we know by conservation of flow to be zero. The incoming flow is subtracted because of the convention that $f(u,v) = -f(v,u)$.

Let's break up the sum another way:

$$\begin{aligned} |f| &= \sum_{u \in S, v \in V} f(u,v) \\ &= \sum_{u \in S, v \in T} f(u,v) + \sum_{u \in S, v \in S} f(u,v) \end{aligned}$$

The first summation on the last line is exactly the flow crossing the cut. The second summation on the last line equals zero, because of the same fact we used above: for every $u, v \in S$, we are summing $f(u,v)$ and $f(v,u)$, and we know that $f(u,v) = -f(v,u)$. Noting that the flow along an edge $f(u,v)$ can be no larger than the capacity of that edge $c(u,v)$, we get, finally:

$$\begin{aligned} |f| &= \sum_{u \in S, v \in V} f(u,v) \\ &= \sum_{u \in S, v \in T} f(u,v) + \sum_{u \in S, v \in S} f(u,v) \\ &= \sum_{u \in S, v \in T} f(u,v) \leq \sum_{u \in S, v \in T} c(u,v) = c(S,T), \end{aligned}$$

which proves part (1). Notice also that we proved along the way that $|f| = \sum_{u \in S, v \in T} f(u,v)$, for any flow f and any cut (S,T); we will use that again in the second part.

Proof of (2). Let f be a maximum flow in G. We will constuct a cut (S,T) such that $c(S,T) = |f|$. We know that there can be no augmenting paths in the residual network G_f. If there were, then f could be augmented and it could not have been a maximum flow. Let S be the set of all vertices reachable from the source s *in the residual network* G_f. This set cannot include the sink t because then there would be an augmenting path. If we let T be the vertices *not* reachable from s in the residual network, then (S,T) is a cut in G.

What can we say about the edges in the original network G, that cross the cut (S,T) in the $s \to t$ direction? Consider such an edge (u,v), where $u \in S$ and $v \in T$. If the flow along edge (u,v) is *less than* the capacity of edge (u,v), then the excess capacity will show up as an edge from u to v in residual graph with non-zero capacity. But that can't be, because by our choice of S and T, u is reachable from s in the residual graph, but v is not! Therefore the flow along edge (u,v) must *equal* its capacity. We say that edge (u,v) is *saturated.*

Since we have just argued that all edges that cross the cut in the $s \to t$ direction are saturated, we have:

$$\sum_{u \in S, v \in T} f(u,v) = \sum_{u \in S, v \in T} c(u,v) = c(S,T).$$

Putting that together with the fact that $|f| = \sum_{u \in S, v \in T} f(u,v)$, which we proved earlier, we have:

$$|f| = \sum_{u \in S, v \in T} f(u,v) = \sum_{u \in S, v \in T} c(u,v) = c(S,T),$$

which is what we set out to prove. ■

Chris Umans
Office Hours: 3:00–5:00 Tuesdays and Thursdays in Soda 583
umans@cs.berkeley.edu *April 12, 2000*

Sections: 10:00–11:00 in Cory 289
11:00–12:00 in Etcheverry 3109

- **The P versus NP question**: The question of whether P equals NP is one of the most important open questions in Computer Science and Math. The P/NP problem is a deep and significant problem; here's part of the reason why: there are at least two *dramatically* different "worlds" we might be living in, and we don't know which one we are in.
  - In World 1, $P \neq NP$. We think we live in this world, but we are not sure. In this world, there are easy problems (ones in P), and many hard problems (NP-complete problems).
  - In World 2, $P = NP$. This is a strange world. Here, designing algorithms is as easy as describing how to recognize a solution. There is (in some sense) a single "super" algorithm, that can be easily adapted to solve almost every problem in CS (like Linear Programming on a much grander scale). And, interestingly, all currently-used cryptography can be broken.
- **Why you should care**: P, NP, and NP-completeness are practically useful concepts as well. Picture an algorithm designer who knows about these concepts versus an algorithm designer who doesn't. The algorithm designer who doesn't know about NP-completeness will probably waste lots of time throughout their career trying to invent efficient algorithms for problems for which such algorithms are probably not possible (some of you experienced this first-hand with the Garage Sale Problem, before the TAs told you it was NP-complete). The algorithm designer who does know about NP-completeness, when given a hard problem, can often prove with relative ease that trying to find an efficient exact algorithm is a waste of time, and go on to more fruitful alternative approaches (such as approximation algorithms).
- **What you need to be able to do**: Given a problem Q_{new}, you should know how to prove it is NP-complete. This involves two steps:
  1. Prove Q_{new} is in NP.
  2. Reduce a known NP-complete problem, Q_{old} to Q_{new}.

A problem Q that is NP-complete, is one of the "hardest" problems in the class NP; moreover, if you can solve Q in polynomial time, then *every problem* in NP can be solved in polynomial time, and hence $P = NP$. Before we describe how to do each step, we need to know exactly what is meant by a "problem."

- **Optimization and decision problems**: For our purposes, a *problem* is a question with the following form:

  1. Given $\underbrace{\qquad}_{\text{instance}}$, what is the shortest/longest/minimum/maximum $\underbrace{\qquad}_{\text{structure}}$?
     E.g., Given a graph G, what is the traveling salesperson tour of minimum length?
  2. Given $\underbrace{\qquad}_{\text{instance}}$, is there a $\underbrace{\qquad}_{\text{structure}}$?
     E.g., Given a graph G *and* an integer k, is there a traveling salesperson tour of length $\leq k$?

  The first type of problem is an *optimization problem.* The answer to the question it asks is some optimal structure (like a shortest tour in the graph). The second type of problem is a *decision problem.* The answer to the question it asks is YES or NO. It is important to remember that P and NP are classes of decision problems. Optimization problems can be transformed into decision problems by adding a "bound" k to the problem instance, as we did above. On your homework you are asked to show that the decision and optimization versions of a problem are "equivalant", up to a polynomial factor.

- **Showing a problem is in NP**: A (decision) problem Q is in NP if there exists a polynomial-time checking algorithm. This algorithm is given an instance I of Q and a "potential solution" C to Q, and it outputs either YES or NO. We require that

  1. If I is a "yes" instance, then there exists some C (a *certificate*) that makes the checking algorithm output YES.
  2. If I is a "no" instance, then no C makes the checking algorithm output YES.

  In practice, you should think in terms of a short certificate you would need to provide to "convince" the checker that a "yes" instance really is a YES instance, together with an efficient procedure for the checker. Some examples (for problems you have seen):

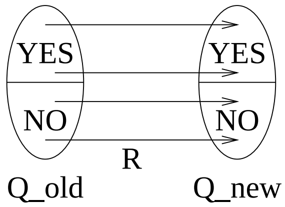
  – 3-SAT: The instance is a 3-CNF Boolean formula; the certificate is a (purported) satisfying assignment. The checker assigns the values specified by the assignment, and checks that each clause contains a TRUE.

– Hamiltonian Path: The instance is a graph G vertices; the certificate is a list of vertices that (supposedly) forms a Hamiltonian path in G. The checker makes sure that every vertex appears exactly once in the list, that there is an edge between each pair of subsequent vertices in the list.

– Maximum Independent Set: The instance is a graph G and an integer k; the certificate is a (supposedly) independent set of the vertices of size k. The checker makes sure no pair of the vertices has an edge between them, and that there are indeed k different vertices in the set.

– Graph Isomorphism: The instance is a pair of graphs G_1 and G_2; the certificate is a (purported) isomorphism between G_1 and G_2 (i.e., a bijective mapping π from the vertices of G_1 to the vertices of G_1). The checker verifies that π is indeed bijective, and that for every edge (u, v) in G_1, the edge $(\pi(u), \pi(v))$ belongs to G_2, and that for every edge (u, v) in G_2, the edge $(\pi^{-1}(u), \pi^{-1}(v))$ belongs to G_1.

- **Reductions**: A polynomial-time reduction from a decision problem Q_{old} to Q_{new} is a polynomial-time algorithm R that is given an (arbitrary) instance of Q_{old} and produces an instance of Q_{new}. We require that "yes" instances of Q_{old} map to "yes" instances of Q_{new}, and that "no" instances of Q_{old} map to "no" instances of Q_{new}. Graphically, the mapping looks like this:



To remember which direction the reduction goes in, it is useful to remember our purpose in performing the reduction in the first place. A reduction from Q_{old} to Q_{new} allows us to argue: "If Q_{new} has a polynomial-time algorithm, then Q_{old} does." Why is this true? Suppose we had an efficient algorithm A for Q_{new}. Then given any instance I of Q_{old}, we could apply the reduction R, obtaining an instance $R(I)$ of Q_{new}, and solve that instance using A. Because yes maps to yes and no maps to no, we know that the yes/no answer for $R(I)$ is also the yes/no answer for I. If you are not sure if you have reduced in the right direction, try making this argument, and see if it works.

- **A sample reduction**: Let's try reducing 3-SAT to Independent Set. Here are the two (decision) problem definitions:

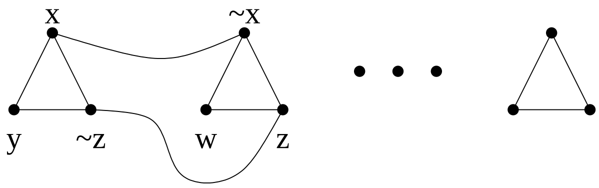
– 3-SAT: Given a Boolean formula in 3-CNF form (an AND or ORs, where each OR involves at most 3 literals), is there a satisfying assignment?

– Independent Set: Given a graph $G = (V, E)$ and an integer k, is there a independent set $V' \subseteq V$ of size at least k? (An independent set is a set of vertices, no two of which are connected by an edge).

Our reduction takes an instance of $3 - SAT$, a 3-CNF ϕ with m clauses, which looks something like:

$$(x \vee y \vee \neg z) \wedge (\neg x \vee w \vee z) \wedge \cdots \wedge (\cdots)$$

and produces a graph G and an integer k. The graph looks like this:



In general, there will be a triangle for each of the m clauses (or an edge or a single vertex if there are only 2 or 1 literals in the clause). To help us reason about the problem we label the vertices of the triangle with the literals in the corresponding clause. Then we connect every pair of vertices labeled with contradictory literals. Finally we set $k = m$.

To show that "yes maps to yes and no maps to no," we prove the following claim. Notice that we need to prove both directions of this claim, even though the reduction is only a mapping in one direction.

Claim 2 *ϕ has a satisfying assignment if and only if G has an independent set of size at least k.*

Proof: ($\rightarrow$) Suppose ϕ has a satisfying assignment. Then under that satisfying assignment there must be at least one TRUE in each clause. Pick one TRUE literal in each clause, and include the corresponding vertex in the independent set. This set has size $m \geq k$, and it is in fact an independent set because we cannot have included any vertices labeled with contradictory literals, since the TRUES came from a truth assignment.

($\leftarrow$) Suppose G has an independent set V' of size at least k. No more than one vertex from each triangle can be in V', so V' must have size exactly $k = m$. Furthermore, no two vertices labeled with contradictory literals can be in V' because it is an independent set. Then, we can set the literals labeling vertices in V' to TRUE in a consistent way, and set the other variables in any way we wish, and we will have at least one TRUE literal in each clause, which is a satsifying assignment. ■

Chris Umans *April 19, 2000*
Office Hours: 3:00–5:00 Tuesdays and Thursdays in Soda 583
`umans@cs.berkeley.edu`

Sections: 10:00–11:00 in Cory 289
11:00–12:00 in Etcheverry 3109

- **Proving 0/1 integer programming NP-complete**: This was the short exercise. The 0/1 integer programming problem is defined in CLR as follows:

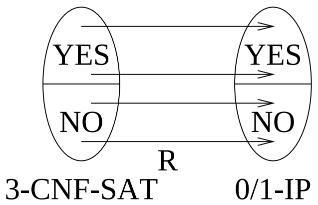
  – 0/1-IP: Given an integer $m \times n$ matrix A and an integer m-vector $\vec{b}$, is there an 0/1 n-vector $\vec{x}$ such that $A\vec{x} \leq \vec{b}$?

  Recall that there are two steps in proving a problem is NP-complete. First, we show that 0/1-IP is in NP. We ask ourselves, "For a yes instance, what short certificate would convince a efficient verifier that this is indeed a yes instance?" Clearly, the solution vector $\vec{x}$ would suffice. The checking procedure takes an instance A, b, and a 0/1 vector $\vec{x}$ that is a (purported) solution, and checks that $A\vec{x} \leq \vec{b}$. As this involves checking m inequalities, it can be done easily in polynomial time. This shows that 0/1 IP is in NP.

  Now we follow the hint and reduce from 3-CNF-SAT to 0/1-IP. Recall the definition of 3-CNF-SAT:

  – 3-CNF-SAT: Given an Boolean formula ϕ that is the AND or ORs, with each OR involving at most 3 literals, is there a satisfying assignment for ϕ?

  The reduction R needs to perform the following mapping:



  That is, given an (arbitrary) instance of 3-CNF-SAT, ϕ, which looks something like:

$$(z_1 \vee z_2 \vee \neg z_3) \wedge (\neg z_1 \vee z_4 \vee z_3) \wedge \cdots \wedge (\cdots),$$

  we need to produce an instance of 0/1-IP, (which consists of m inequalities involving n 0/1 variables) specified by an $m \times n$ matrix A and an m-vector $\vec{b}$.

  Before attempting the transformation, let's remind ourselves of the claim we will need to be able to prove:

Claim 3 *ϕ has a satisfying assignment if and only if there exists a 0/1 vector $\vec{x}$ for which $A\vec{x} \leq \vec{b}$.*

Since the 3-CNF-SAT instance involved finding a truth assignment to the z variables, it seems natural to have the x variables of the IP correspond to the z variables. So, we set n to be the number of z variables in ϕ, and we have one 0/1 x variable for each in our instance of 0/1 IP. We will associate the 0/1 value of variable x_i in the IP with the FALSE and TRUE values of the corresponding z_i variable in ϕ.

Now, we need to constrain the x variables so that there is a solution to $Ax \leq b$ if and only if there is a satisfying assignment to ϕ. Notice that the IP solution must satisfy ALL of the inequalities, just as a satisfying assignment must satisfy ALL of the clauses of ϕ. This suggests that we should have a constraint in our IP for each clause in ϕ. For a clause in ϕ with no negated variables, like $(z_1 \vee z_2 \vee z_3)$, it is pretty clear that we want a constraint like:

$$x_1 + x_2 + x_3 \geq 1$$

Thinking of our correspondence between IP variables and variables of ϕ, this constraint says "at least one variable in the clause must be true."

What constraint should we write for a clause like $(z_1 \vee z_2 \vee \neg z_3)$? We need a (linear!) function f that behaves like $\neg$. That is, we would like $f(x_3) = 0$ if $x_3 = 1$ and $f(x_3) = 1$ if $x_3 = 0$. It seems that $f(x_3) = 1 - x_3$ will do the trick. Then we can encode a clause like $(z_1 \vee z_2 \vee \neg z_3)$ with a constraint like:

$$x_1 + x_2 + (1 - x_3) \geq 1.$$

We now have all we need to specify the reduction. We set m to be the number of clauses in ϕ, and for each clause, we write the corresponding constraint on the x variables. Using the standard algebraic manipulation we can get all of the constraints into the form $a_1x_1 + a_2x_2 + \cdots a_nx_n \leq b_j$, where the a_i and b_j are constants. From there we get the matrix A and the vector $\vec{b}$. Finally, we can prove the claim above:

Proof:[of claim] ($\rightarrow$) Assume ϕ has a satsifying assignment. Then for each i we can set the IP variable x_i to 1 if z_i is TRUE in the satisfying assignment, or 0 if z_i is FALSE. Since the satsifying assignment has at least one TRUE in each clause, there is at least one x_i set to 1 (or $(1 - x_i)$ set to 1) in each constraint of the IP; hence we have described a vector $\vec{x}$ for which $A\vec{x} \leq \vec{b}$.

($\leftarrow$) Assume there is some vector $\vec{x}$ for which $A\vec{x} \leq \vec{b}$. Then for each i, set the variable z_i to TRUE if x_i is set to 1 and FALSE if x_i is set to 0. Since every IP constraint is obeyed, every clause in ϕ has a TRUE in it. We have exhibited a satisfying assignment and therefore ϕ is satisfiable. ■

- k**-coloring example**: As further practice with reductions, consider the 3-coloring problem (which you will prove to be NP-complete in a homework):

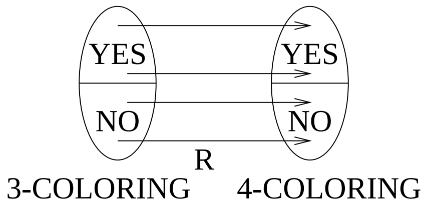
  - 3-Coloring: Given a graph G, is there a coloring of the vertices with 3 colors so that no two adjacent vertices are colored with the same color?

  We will try to prove that the 4-coloring problem is NP-complete:

  - 4-Coloring: Given a graph G, is there a coloring of the vertices with 4 colors so that no two adjacent vertices are colored with the same color?

  We will skip over verifying that 4-coloring is in NP; it should be quite easy.

  We now want a reduction R that performs the following mapping:



  We are given an (arbitrary) graph G, and we want the reduction to produce a graph G' so that we can prove the following claim:

  Claim 4 *G is 3-colorable if and only if G' is 4-colorable.*

  As a first attempt, we might try setting $G' = G$. One direction of the claim works – if G is 3-colorable, then $G' = G$ is certainly 4-colorable (we can just not use one color). However, the other direction is problematic. If G' is 4-colorable, we don't know that G is 3-colorable. We need to find a way to "force" G' to "waste" one color, so that the remaining coloring is essentially a 3-coloring of G. We can do this by defining G' to be G with one extra vertex v that is connected to every other vertex. Having specified our reduction, we can now prove the claim:

  Proof:[of claim] ($\rightarrow$) Suppose we have a 3-coloring of G. Then we can use the same colors to color all of G' except the extra vertex v with 3 colors, and then color v with the 4-th color. Hence G' is 4-colorable.

($\leftarrow$) Suppose we have a 4-coloring of G'. Notice that whatever color is used by the extra vertex v *cannot* be used by any of the other vertices (because v is connected to every other vertex). Therefore the remaining vertices of G' (which are exactly the vertices of G) must be colored with only 3 colors, which implies that G is 3-colorable. ■

Notice that we can generalize this reduction. We can prove the problem k-Coloring, for any $k \geq 4$, is NP-complete (e.g., 29-Coloring is NP-complete). We use the same idea, but now we need to add $k-3$ extra vertices, each of which is connected to every other vertex (including the other $k-4$ extra vertices).

Chris Umans *April 27, 2000*
Office Hours: 3:00–5:00 Tuesdays and Thursdays in Soda 583
umans@cs.berkeley.edu

Sections: 10:00–11:00 in Cory 289
11:00–12:00 in Etcheverry 3109

- **Solving a modular linear equation, example:** The short exercise asked you to solve the following equation:

$$35x \equiv 10 \pmod{50}$$

By the theorem from lecture, this equation has a solution if and only if $\gcd(35, 50)$ divides 10. We should check that first. By inspection, $d = \gcd(35, 50) = 5$, (we could have computed it using Euclid's algorithm), and 5 divides 10, so the equation has a solution. Also by the theorem, it has $d = 5$ distinct solutions modulo 50. To find the solutions, we run ExtendedEuclid on 50 and 35:

$$\begin{array}{ccccccc} EE(50,35) & \rightarrow & EE(35,15) & \rightarrow & EE(15,5) & \rightarrow & EE(5,0) \\ & & & & & & \downarrow \\ (5,-2,3) & \leftarrow & (5,1,-2) & \leftarrow & (5,0,1) & \leftarrow & (5,1,0) \end{array}$$

The top row (from left to right) is meant to indicate the series of recursive calls to ExtendedEuclid (EE); we use the fact that $EE(a, b)$ calls $EE(b, a \bmod b)$. The bottom row (from right to left) shows the triplet returned by each call; we use the fact that if $EE(b, a \bmod b)$ returns (d, x', y') then $EE(a, b)$ returns $(d, y', x' - \lfloor a/b \rfloor y')$.

The result gives us a linear combination of 50 and 35 that equals 5:

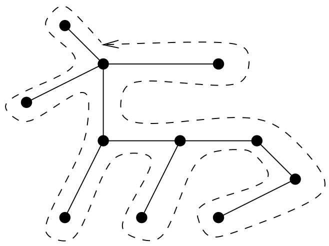
$$(3)(35) + (-2)(50) = 5.$$

Now we consider this equation modulo 50:

$$\begin{aligned} (3)(35) + (-2)(50) &\equiv 5 \pmod{50} \\ (3)(35) &\equiv 5 \pmod{50} \\ (6)(35) &\equiv 10 \pmod{50} \end{aligned}$$

The last line is obtained by multiplying both sides by 2. Now, we have a solution to the equation, namely, 6. The theorem tells us that the other 4 distinct solutions (modulo 50) can be obtained by adding multiples of $50/d = 50/5 = 10$ to this solution. The 5 distinct solution modulo 50 are therefore $6, 16, 26, 36, 46$.

- **Christofides' algorithm**: We discussed an approximation algorithm for TSP with the triangle inequality in lecture; here is a more clever variant. Recall the approximation algorithm from class: we are given a complete graph G whose edge weights satisfy the triangle inequality. We find a minimum spanning tree M in G, and form a tour T' by going "twice around" this tree, as pictured below:

  We then transformed tour T' into tour T by "short-circuiting" the tour so that it visited every vertex exactly once. That is, starting at the root, we followed T' until it was about to visit an already-visited vertex, at which point we moved directly to the next unvisited vertex on tour T'.

  We noticed that because an optimum traveling salesperson tour of G yields a spanning tree with the same or smaller cost (by simply deleting an edge from the tour), that $w(M) \leq w(OPT)$. Here $w(M)$ is the weight of spanning tree M, and $w(OPT)$ is the length of the optimum traveling salesperson tour.

  Because of the triangle inequality, the short-circuiting process cannot have increased the tour length, so we have:

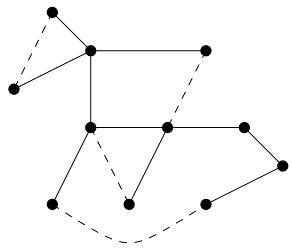
  $$w(T) \leq w(T) = 2w(M) \leq 2w(OPT),$$

  which means that we have an approximation algorithm with ratio 2, since it returns a tour that is no longer than twice the optimum.

  Christofides' algorithm introduces a clever twist, and improves the ratio to 1.5. It begins in the same way, by finding in $G = (V, E)$ a minimum spanning tree M.

  It then finds a *minimum weight perfect matching* on V', the odd-degree vertices of M. This requires some explanation. First, notice that M (indeed any graph) has an even number of odd-degree vertices. This is because the sum of the degrees equals twice the number of edges, an even number. Now, a *perfect matching* is simply a matching in which every vertex is touched by an edge in the matching. The weight of a matching is just the sum of the weights of the edges in the matching. A minimum weight perfect matching (MWPM) is a perfect matching with minimum weight. There is a polynomial-time algorithm that finds a minimum weight perfect matching, but we won't describe it here.

Let us call the MWPM we find P. We add the edges of P to our MST M. In our example graph, we might get something like this:



Notice that in the resulting graph, every vertex has even degree! We have "repaired" all of the odd-degree vertices in V' by matching them with other odd-degree vertices. What do we know about graphs in which every vertex has even degree? There is an Euler circuit (a tour that traverses every edges exactly once and returns to its starting point). We let T' be an Euler circuit, and then short-circuit it to get an honest traveling salesperson tour T as before.

Now, how long is our tour T? We know that $w(T) \leq w(T')$ as before, and that $w(T') = w(M) + w(P)$, since T' traverses every edge in the spanning tree M and the perfect matching P exactly once. As before we have the inequality $w(M) \leq w(OPT)$. We need to know the relationship between $w(OPT)$ and $w(P)$.

We claim that $2w(P) \leq w(OPT)$. This is not hard to see: first, notice that an optimum traveling salesperson tour on *only* the vertices in V' (call it OPT') can be no longer than OPT, by the triangle inequality – we could just short-circuit OPT to skip over the vertices not in V', making the tour no longer in the process. Now, since there are an even number of vertices in V', OPT' is a cycle of even length. Therefore, by taking every-other edge around OPT', we get a perfect matching on V', and by taking the other edges around OPT' we get another perfect matching on V'. Each of these perfect matchings can have weight no larger than the MWPM, so we have:

$$w(OPT) \geq w(OPT') \geq 2w(P).$$

This implies that $w(P) \leq 0.5w(OPT)$. Putting everything together, we can bound the length of our tour T:

$$w(T) \leq w(T') = w(M) + w(P) \leq w(OPT) + 0.5w(OPT) = 1.5w(OPT).$$

Therefore, the algorithm we just described attains an approximation ratio of 1.5, since it returns a tour that is no longer than 1.5 times the optimum.

Chris Umans *May 3, 2000*
Office Hours: 3:00–5:00 Tuesdays and Thursdays in Soda 583
umans@cs.berkeley.edu

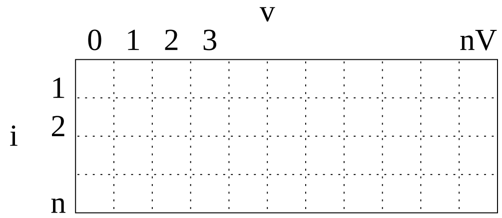
Sections: 10:00–11:00 in Cory 289
11:00–12:00 in Etcheverry 3109

- **A pseudo-polynomial-time algorithm for knapsack**: The Knapsack optimization problem is defined as follows:

  – Given n elements with integer values $v_1, v_2, \ldots v_n$ and weights $w_1, w_2, \ldots w_n$, and a capacity W, what is the collection of elements with maximum total value, whose weight does not exceed W?

  The decision version of this problem is NP-complete (it should seem similar to Subset-Sum and Bin-Packing), but we will not prove that here. Instead we will focus on a dynamic programming algorithm that produces an *exact* solution to this problem.

  We first describe the subproblems and the dynamic programming table. Define $K(i, v)$ to be the minimum weight of a subset of the first i elements with total value v (or ∞ if no such subset exists). Let us denote by V the maximum value of any element; i.e., $V = \max_i v_i$. The dynamic programming table looks something like this:



  We know that nV is an upper bound on the maximum possible value attainable, so it is safe to allocate only nV columns in the table. To figure out the solution to the knapsack problem from this table (once all the cells are filled in), we consider the values in the bottom row from right to left. Starting in column nV, if the bottommost cell has weight $\leq W$, then we have a solution with value nV. If not, there is no way to select items with value nV and permissible weight. So we check if the weight in cell $(n, nV - 1)$ is $\leq W$. If so, we have a solution with value $nV - 1$. We continue in this fashion until we see a cell (n, v) with weight $\leq W$; this v is then the solution to the problem.

To fill in the table, we use the following recurrence:

$$K(i,v) = \min\{\underbrace{K(i-1,v)}_{\text{exclude item } i}, \underbrace{K(i-1,v-v_i)+w_i}_{\text{include item } i}\}.$$

$$K(1,v) = \begin{cases} w_1 & \text{if } v_1 = v \\ \infty & \text{otherwise} \end{cases}$$

The running time of this algorithm is $O(n \cdot nV) = O(n^2V)$. At first glance this seems like a perfectly good efficient algorithm for a problem we said was NP-complete. Why does this not show $P = NP$? The running time depends on V, the maximum value of the elements in the problem. This value might be exponentially large, because we can represent a value of size about 2^n using n bits in the input to the problem. However, for "reasonable" values for the elements, this solution works well. This is what is known as a *pseudo-polynomial* time algorithm, because its running time depends polynomially on the *magnitude* of the values in the problem instance, instead of polynomially on the size of their representation.

- **A polynomial-time approximation scheme**: The algorithm we just described can be adapted to give an approximation algorithm for knapsack that is the best sort of approximation algorithm we can hope for: for *any* given $\epsilon > 0$, we will be able to devise a polynomial-time algorithm that gets ϵ-close to the optimum.

  Let's consider what a "bad" case for the exact algorithm above might look like. We might have really large values for the elements, like this:

$$\begin{aligned} v_1 &= 10000034 \\ v_2 &= 28000001 \\ v_3 &= 34000110 \\ v_4 &= 13500003 \end{aligned}$$

  Since we only need an approximate solution, a good idea would be to truncate these values, keeping only the most significant digits, and solve the resulting knapsack problem, which now has smaller values.

  Let's make this idea more formal: define a new instance of knapsack from the old one by truncating the last k bits of every value. That is, $v_i' = \lceil v_i/2^k \rceil$ for each i, and then the new maximum value $V' = \lceil V/2^k \rceil$. Now, let's run the exact dynamic programming algorithm on this new instance, and then multiply the value it achieves by 2^k at the end. We'll call the value output by this procedure $APPROX$.

We notice that the feasible subsets of items (subsets whose total weight does not exceed W) remains unchanged, so our algorithm's solution is a valid solution to the original instance. Because each element potentially lost some value in the scaling down/scaling up process, $APPROX \leq OPT$, where OPT is the value of an optimum solution to the original problem. But how much smaller that OPT can $APPROX$ really be?

We claim that each item lost at most 2^k in value (because we essentially ignored the lower order k bits). Since there are n items, we have:

$$APPROX \geq OPT - n2^k.$$

We then have:

$$\frac{APPROX}{OPT} \geq \frac{OPT - n2^k}{OPT} = 1 - \frac{n2^k}{OPT} \geq 1 - \frac{n2^k}{V},$$

where the last inequality is true because $OPT \geq V$ (we can always include *only* the highest-value element).

So now, if we want to get ϵ-close to OPT, we set $k = \log_2(\epsilon V/n)$. Plugging in, we get:

$$\frac{APPROX}{OPT} \geq 1 - \frac{n2^k}{V} = (1 - \epsilon)OPT.$$

The running time of the approximation algorithm is:

$$O(n^2 V') = O(n^2 \lceil V/2^k \rceil) = O(n^3/\epsilon),$$

which is polynomial for any given choice of $\epsilon > 0$.

What we have described is a *polynomial-time approximation scheme*, or "PTAS". A PTAS is a *class* of approximation algorithms, one for any $\epsilon > 0$ that gets ϵ-close to the optimum. Each algorithm in the class runs in polynomial time. The technique of truncating values that we used here can be more generally applied to problems for which there is a pseudo-polynomial time exact algorithm.