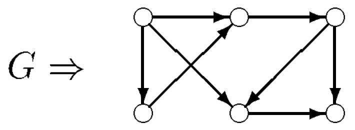
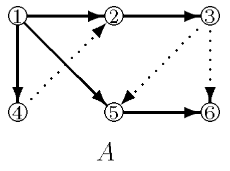
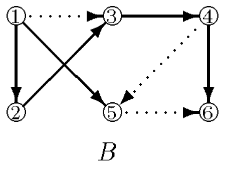
CS-170 Solution to Exam 2 March 22, 1995
David Wolfe

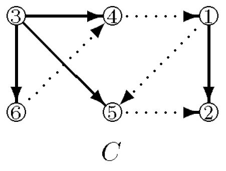
1. (7 points per part) **No words are required (nor recommended).** Consider the following graph G:



Consider graphs A, B and C below (shown as **solid** edges), all subgraphs of G:







Some answers to parts a,b, and c below may be the same. Vertex numbers are for part (c) only.

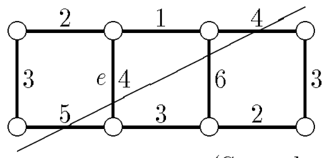
(a) Which **one** of the subgraphs indicated could be a breadth first search tree? $\boxed{\text{A}}$

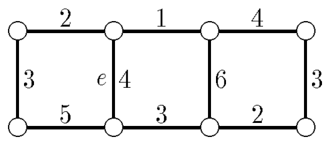
(b) Which **one** of the subgraphs could be a depth first search forest? $\boxed{\text{A}}$

(c) Which **one** of the graphs is labeled with a topological sort <u>of G</u>? $\boxed{\text{B}}$

(d) How many strongly connected components does original graph, G, have? $\boxed{6}$

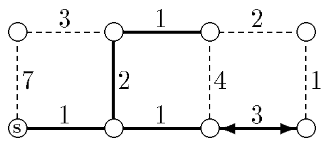
(e) For the following graph, prove edge e is part of **some** minimum spanning tree by drawing an appropriate cut in the graph. (Just draw the cut.)

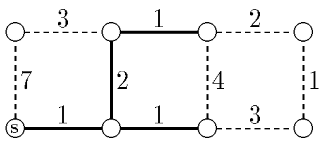




(Second copy is for your convenience.)

(f) Dijkstra's algorithm is being run to find shortest paths from source s in the following undirected graph. The graph is shown in dashed edges, and the tree so far is solid. Darken the dashed edge which would next be added to the tree.





(Second copy is for your convenience.)

2. (20 points) The all-pairs *reachability* problem: Given a directed graph $G = (V, E)$ in the form of an **adjacency matrix**, determine for all pairs of vertices $u, v \in V$ if there is a path from u to v. Propose an efficient algorithm to solve this problem. You'll receive:

- full-credit for $O(V(E + V))$
- extra-credit for anything faster, say $O(V^2)$ or $O(V^2 \log V)$

Convert the adjacency matrix to an adjacency list representation in $O(V^2)$ time, and construct a depth-first search tree from each vertex u to see which vertices it can reach — $O(V(E + V))$.

3. (30 points) Consider the following variant of the 0-1-knapsack problem in which there is an unlimited supply of each item. You have won a shopping spree at a store where there are n products; the i^{th} has size w_i and value v_i. You have a grocery cart which can be filled with products whose size totals to W, taking as many of each product as you wish. Your goal is to maximize the total value of the products you take.

   (In the 0-1 knapsack problem, you take at most one of each product. In this problem, there is an unbounded supply of each product.)

   Determine two dynamic programming algorithms to the value of goods, V, which you can take during your shopping spree. It suffices to give a recurrence with a one line explanation verifying the running time of a dynamic programming algorithm to solve the recurrence.

   (a) Give an $O(nW)$ solution. (Hint: Let V_w be maximum value you can pack into a grocery cart holding total size w.)

   (b) Give an $O(nV)$ solution, where V is the total value you can pack in the grocery cart in the solution. (Hint: Let W_v be the minimum sized basket you need to hold a total value of v.)

(By the way, an $O(nW^2)$ solution duplicates each item W times, and runs 0-1 knapsack as a subroutine.)

For an $O(nW)$ solution,

$$V_w = \max\left(0, \max_{1\le i\le n} v_i + V_{w-w_i}\right)$$

Each V_w takes $O(n)$ time to compute, so computing $V_0, \ldots, V_W$ takes $O(nW)$ time.

For an $O(nV)$ solution,

$$\begin{aligned} W_v &= \min_{1\le i\le n} W_{v-v_i} + w_i \\ W_v &= 0 \qquad \text{(For } v \le 0\text{)} \end{aligned}$$

If $W_{v+1} > W$ but $W_v \le W$, then $V = v$ is the solution. Computing each W_v takes $O(n)$ time, so computing $W_0, \ldots, W_{V+1}$ takes $O(nV)$ time.

4. (30 points) Give the best algorithm you can to find the k^{th}-smallest element in an n-node binary heap, where n is **much** larger than k.

A trivial lower bound is $\Omega(k)$: $\sim k/2$ elements must be examined, since the minimum could be any of the $\sim k/2$ elements at depth $\lg k$. Frederickson (22nd STOC, 1990) showed that this bound is tight, contradicting a fallacious lower bound paper of $\Omega(n \lg n)$, as well as the intuition of many theoreticians.

A trivial upper bound is $O(k \lg n)$ by simply doing k DELETE-MIN's on the heap. Another algorithm works in $O(n)$ time by ignoring the heap and using linear time selection. The selection algorithm is found in CLR 10.3 and is a fundamental algorithm which you should understand.

For an $O(2^k)$ algorithm, note that only the top k levels (containing $\Theta(2^k)$ nodes) can contain the k^{th} smallest. So just cut the entire heap after this depth in time $O(2^k)$, and run k DELETE-MINs taking time $O(k\log(2^k)) = O(k^2)$.

For an $O(k^2)$ algorithm, we don't actually have to cut the off the bottom of the heap; instead just act as though it doesn't exist when doing the DELETE-MINs.

Now for an $O(k \lg k)$ algorithm. The key is to take advantage of the heap order without doing DELETE-MINs. In particular, the i^{th} smallest element is a child of one of the $i-1$ smallest elements. Initialize an auxilary heap A to contain the minimum element in the original heap, H. At stage i, do $x_i \leftarrow$ EXTRACT-MIN(A), and insert x_i's children in H into A. Return x_k after doing $2k-1$ inserts and k EXTRACT-MIN's on the auxilary heap.