CSC 477/677 Spring 2022 Final Examination 1:00-3:00 May 9, 2022

Topics to Study

1. Asymptotic Classes of Functions
   - (i) O, Ω, and Θ. What do those mean?
   - (ii) Logarithmic and Polylogarithmic What do those mean?
   - (iii) Polynomial What is a polynomially bounded function? What is a $\mathcal{P}$–TIME computation? $\mathcal{P}$–SPACE computation?
   - (iv) Exponential
2. Time Complexity and Asymptotic Time Complexity of Programs
   - (i) Complexity of Algorithm we've Learned
   - (ii) Recurrences
3. Computational Paradigmns
   - (i) What is a greedy algorithm?
   - (ii) What is a dynamic programming algorithm?
     Relationship with directed acyclic graphs
   - (iii) What is a divide and conquer algorithm? Relationship with recursion
4. Sorting
   - (i) Selection sort.
   - (ii) Insertion sort.
   - (iii) Margesort. Divide and Conquer. Start with an array X. Let X_1 be the left half and X_l the right half. That takes O(1) time. Then use mergesort recursively to sort X_1 and X_2. Then merge them into X, which takes $O(n)$ time. The recurrence for the time complexity is $T(n) = 2T(n/2) + n$.
   - (iv) Quicksort. Dual to mergesort. Start with an array X. Select one entry of X to be **pivot** Partition X into X_1 and X_2 such that all items of X_1 are no greater than **pivot**, and items of X_2 are no smaller than pivot. Sort X_1 and X_2 recursively, then set X to be the concatenation of X_1 and X_2. The worst case occurs whtn the pivot is the smallest or the largest. The worst case time complexity is $O(n^2)$. The expected time is $O(n \log n)$ if pivots are picked at random.
   - (v) Heapsort. A form of selection sort. First heapify the array. Then select the greatest (or least) element repeatedly.
   - (vi) Treesort. A form of insertion sort. Insert item one at a time into a binary search tree. and then read them out in left-to-right order, also called inorder.
   - (vii) Theoretical $\Omega(n \log n)$ bound. If an algorithm uses comparison between two items to make a decision, that is, the flow chart is a binary tree, the number of comparisons is $\Omega(n \log n)$ in the worst case. The worst-case number of comparisons is absolutely (not asymptotically) bounded below by $\log_2 n!$.

(viii) Radix, or Bucket, sort. Does not use the comparison model, so the $\Omega(n \log n)$ bound does not hold.

5. Searching

   (i) Linear search. Surprisingly, it is optimal in many practical situations, such as separate chaining.

   (ii) Binary search. This is a divide-and-conquer algorithm. The list must be sorted for it to work.

6. Selection

   (i) Median of medians. $\Theta(n)$ time, but a large constant. What is the recurrence?

   (ii) Randomized selection. $O(n)$ expected time. $O(n^2)$ time in the worst case, which in practice "never" occurs.

   Once you divide the file into two parts, you reduce your search space to one of those parts. How do you determine which one?

7. Graphs

   (i) BFS and DFS

   (ii) Vertices and edges

   (iii) Neighbor list

   (iv) Subgraphs

   (v) Connected graphs

   (vi) Components

   (vii) Acyclic graphs

   (viii) Planar graphs, Euler's formula: $m \leq 3n - 6$

   (ix) Complete graphs

   (x) Trees
   A graph is a tree if it is acyclic and connected.

   (xi) Weighted graphs, minimum spanning trees

   Directed graphs = digraphs

   (i) Vertices and arcs

   (ii) In-neighbors, out-neighbors

   (iii) Subgraphs

   (iv) BFS and DFS

   (v) Strongly connected digraphs

   (vi) Strong components

   (vii) Acyclic digraph = dag (very different from acyclic graphs)

   (viii) Topological order of a dag

   (ix) Trees (not the same definition as above)

   (x) Transitive closure and transitive reduction of a digraph (we never covered this)

(xi) Weighted digraphs and shortest path problems

   i. Single pair.
   ii. Single source.
   iii. All pairs.
   iv. Bellman-Ford algorithm.
   v. Floyd-Warshall algorithm.
   vi. Dijkstra's algorithm.
   vii. Johnson's algorithm.
   viii. A^* algorithm.

8. Search Structures A search structure holds items, and can be searched for a specific item. Items can be inserted or deleted.

   (i) Unordered Lists.

      i. Move to Front.

   (ii) Trees.

      i. Binary Search Tree. Insert the letters A,Q,F,Z,T,L,R into a binary search tree, in that order. Show the final tree.
      ii. B-tree. We never went over that, so it won't be on the final.
      iii. Treap. Insert the sae sequence of items, as in (a), where each has a "random" key given in the following table.

   (iii) Hash Tables.

      i. Collisions
      ii. Closed hashing, open addressing.
      iii. Open hashing, separate chaining.
      iv. Cuckoo hashing.
      v. Perfect hashing.
      vi. Probe sequence.

9. Priority Queues A priority queue holds a number of items, one of which is the item of highest priority. Any item can be inserted into a priority queue, but only the highest priority item can be deleted.

   Typically, the items in a priority queue represent unfulfilled obligations, and deletion corresponds to fulfilling that obligation.

   (i) Stack. The item of highest priority is the most recently inserted item. Insertion of an item is usually called *push*, and deletion of an item is usually called *pop*. A stack is usually implemented as a list ordered by priority, that is, the time the item was inserted.

      Stacks are the most important priority queues.

   (ii) Queue. The item of highest priority is the least recently inserted item. Insertion of an item is sometimes called *enqueue*, and deletion of an item is sometimes called *dequeue*. A queue is usually implemented as a list ordered by priority, that is, the time the item was inserted.

      My favorite two implementations of a queue are

        i. An array of fixed length where the front and rear are moving indices in the array. In C++ you could use vectors.
        ii. A circular linked list, with a pointer to a dummy node.

    (iii) Heap. The item of highest priority is the item which has the largest or smallest value of some key, such as size, weight, or whatever. The heap is said to be a maxheap or minheap, if priority item has the largest or smallest key, respectively. In a pure heap, the key cannot be altered while it is in the heap, but there are variations where the key can be changed, causing the heap to be reorganized. For example. in Dijkstra's algorithm, the heap is a minheap, and the key of an item in the heap can be decreased.

    There is more than one way to implement a heap, but I only showed one of those in class. The heap is implemented as an almost complete binary tree, in turn implemented as an array, where the tree nodes are stored in level order.

10. Arrays

    (i) Fetch and store
    (ii) Row-major and column-major order
    (iii) Storing a multi-dimensional array in main memory
    (iv) Triangular and ragged arrays
    (v) Sparse arrays

    How can you implement a sparse array using a search structure?

11. Dynamic Programming

    Here is something I got from Quora:

    Originally Answered:
    How should I explain what dynamic programming is to a 4-year-old?
    writes down "1+1+1+1+1+1+1+1 =" on a sheet of paper*
    "What's that equal to?"
    counting* "Eight!"
    writes down another "1+" on the left*
    "What about that?"
    quickly* "Nine!" "How'd you know it was nine so fast?"
    "You just added one more"
    "So you didn't need to recount because you remembered there were eight! Dynamic Programming is just a fancy way to say 'remembering stuff to save time later'"

    Dynamic programming is one of the most important things you've learned in the class. Although very few students were able to work the first problem in Assignment 7, that problem was quite straightforward, meaning that identifying the subproblems was far easier than most dynamic programming problems. Here is the outline of how to solve a dynamic programming problem.

    (i) Identify subproblems.
    (ii) Work the subproblems in topological order. You need to compute each subproblem using the results of earlier subproblems.

(iii) Memoization, an alternative form of dynamic programming, where the order in which you work the subproblems is determined on the fly rather than being decided in advance. Frequently, many of the subproblems will not have to be worked at all. The overhead cost of using memoization is only justified if most of the subproblems do not have to be worked.

If I give you a dynamic programming problem to solve (other than Levenstein distance) I will tell you what the subproblems are: the rest is up to you.

Examples:

(i) Given a sequence X = x[1], x[2] ... x[n] of positive numbers, find a monotone increasing subsequence of maximum length. To avoid ties, we assume that the $x[i]$ are distinct.

There are n subproblems. Subroblem i is to find the maximal length monotone subsequence of X which ends at x[i].

We first introduce the fictitious 0^{th} term, $x[0] = 0$. Recall that $x[i] > 0$ for all $i \geq 1$.

We let M be the maximum length of any monotone subsequence of X, and we let last be the index of the last term of that subsequence.

We let L[i] be the maximum length of any monotone subsequence of X which ends at x[i]. We let B[i] be the backpointer, that is, the index of the secon-to-the-last term of the longest monotone subsequence of X which ends at x[i]. If x[i] is the only term of that subsequence, we let $B[i] = 0$.

Here is the algorithm.

```
dynamicprogram
 {
  x[0] = 0;
  L[0] = 0;
  M = 0;
  last = 0;
  for i = 1 to n
   {
    L[i] = 1;
    B[i] = 0;
    for j = i-1 downto 1
     if(x[j] < x[i] and L[j] >= L[i])
      {
       L[i] = L[j]+1;
       B[i] = j;
       if(L[i] > M)
        {
         last = i;
         M = L[i];
        }
      }
   }
  write "The longest monotone subsequence has length" M
 }
```

We then use the backpointers to find the maximum monotone increasing subsequence, which ends at x[last].

```
writesequence(k)
 {
  if(k > 0)
   {
    writesequence(B[k]);
    write(k)
   }
 }


main
 {
   writesequence(last);
 }
```

You can also run the algorithm backwards. In that case, L[i] is the longest monotone subsequence that starts at x[i], and we use forward pointers instead of backpointers.

(ii) Rod Cutting Problem. `https://www.techiedelight.com/rod-cutting/`

(iii) Subset sum problem. Given a sequence of positive integers $X = x_1, x_2, \ldots x_n$ and another integer K, does there exist a subsequence of X whose sum is K? There is an $O(nK)$ time dynamic programming algorithm which answers this question.[1]

There are nK subproblems. For $i \leq n$ and $M \leq K$, Boolean $S[i, M]$ is true if and only if there is a subsequence of $x_1, \ldots x_i$ whose sum is M. Here is the algorithm:

```
for(int M = 0 to K)
  S[0,M] = 0 // we use 0 to mean false
for(int i = 1 to n)
  for(int M = 0 to K)
    S[i,M] = S[i-1,M] or S[i-1,M-x[i]];
if(S[n,K]) write "Yes";
  else write "No";
```

This algorithm is correct. However, it is not $\mathcal{P}$–TIME in general, but only in special cases. Do you see why?

12. Huffman's Algorithm. Find an optimal prefix code for a given weighted alphabet.

13. Loop Invariants

(i) Give a definition of "loop invariant."

(ii) The purpose of the following code is compute $2n$. Write down the loop invariant.

```
int double(int n)
 {
```

[1]Wait a minute! Didn't we learn that the subset sum problem is $\mathcal{NP}$–complete?

```
  // assert (n >= 0)
  int m = n;
  int rslt = 0;
  while(m > 0)
   {
    m = m-1;
    rslt = rslt+2;
   }
  cout << rslt;
 }
```

(iii) The purpose of the following code is compute x to the power of n. Write down the loop invariant.

```
float power(float x, int n)
 {
  // assert(n >= 0);
  float z = 1.0;
  float y = x;
  int m = n;
  while(m > 0)
   {
    if(m%2) z = z*y;
    m = m/2;
    y = y*y;
   }
  return z;
 }
```