1. (30 pts.) Recall that Dijkstra’s algorithm finds shortest paths from $v_1$ to all other vertices by adding edges linking in the closest points. In the graph shown below, each edge is bidirectional; that is, you can travel in either direction on it. **Edges are labeled with upper case letters.** (Two copies of the graph are provided so you can use one as a “worksheet” if you wish.)

(a) List edges in order chosen by algorithm: A J H B C D L F R

(b) At each vertex, give the length of the shortest path from $v_1$ to the vertex. **Indicate which graph has your answer.**

![Graph](image)

2. (25 pts.) Consider the following eight complexity categories (remember $\lg = \log_2$):

$$\Theta(n) \quad \Theta(n^2) \quad \Theta(2^n) \quad \Theta(3^{\lg n}) \quad \Theta(n^{\lg n}) \quad \Theta(n \lg n) \quad \Theta((\sqrt{n} + \ln n)^2) \quad \Theta(2^{n + \lg n}).$$

(a) Which are equal?

$$\Theta(n) = \Theta((\sqrt{n} + \ln n)^2)$$

(b) Arrange the distinct classes in order from slowest growing to fastest growing. In other words, if $\Theta(f(n))$ is to the left of $\Theta(g(n))$, then $f(n) \in o(g(n))$.

$$\Theta(n) \quad \Theta(n \lg n) \quad \Theta(3^{\lg n}) \quad \Theta(2^n) \quad \Theta(n^{\lg n}) \quad \Theta(n \lg n) \quad \Theta(2^n) \quad \Theta(2^{n + \lg n}).$$
3. (30 pts.) The average running time for an algorithm is a nondecreasing function of \( n \) and satisfies \( T(4n) = T(2n) + 2T(n) \) for all \( n > 0 \). Furthermore, \( T(1) = 1 \) and \( T(2) = 3 \).

(a) Determine \( T(2^k) \) as a function of the integer \( k \).

\[ \text{Hint: Set } t_k = T(2^k). \]

Ans. By the hint, \( t_{k+2} = t_{k+1} + 2t_k \), where \( t_0 = 1 \) and \( t_1 = 3 \). Since the roots of \( x^2 = x + 2 \) are \( x = -1 \) and \( x = 2 \), the general solution to the recursion is

\[ t_k = A(-1)^k + B2^k. \]

With \( k = 0, 1 \), we have \( A + B = 1 \) and \( -A + 2B = 3 \). Hence \( B = 4/3 \) and \( A = -1/3 \). Thus \( T(2^k) = (2^{k+2} - (-1)^k)/3 \).

(b) Determine the complexity class of \( T(n) \).

Ans. \( T(n) \in \Theta(n) \) by Theorem B.4.

4. (30 pts.) Suppose we have two sorted lists \( a_1, \ldots, a_n \) and \( b_1, \ldots, b_n \), both of length \( n \), that we want to merge to obtain a sorted list of length \( 2n \), say \( c_1, \ldots, c_{2n} \). To do this, we must decide where the \( a_i \)'s fit among the \( b_j \)'s to produce the \( c \) list. The number of choices for this is \( \binom{2n}{n} \geq 4^n/(2n^{1/2}) \).

Suppose the merge is done comparisons of keys. Using the above information, derive a lower bound for the worst case number of key comparisons that are needed. Explain your reasoning; don’t just give an answer.

Ans. Each comparison allows us the split the possibilities into two parts. The decision tree will be binary and must have at least \( \binom{2n}{n} \) leaves. Since the longest from root to leaf in such a tree is at least the log base 2 of the number of leaves, \( W(n) \geq \lceil \lg \binom{2n}{n} \rceil \). You could leave off the ceiling function. You could also use the lower bound for the binomial coefficient to get

\[ W(n) \geq 2n - \lg 2 - (\lg n)/2. \]

By the way, this is nearly achieved by the merge process in mergesort: It’s worst case number of comparisons is \( 2n - 1 \).
5. (30 pts.) Here is an informal description of a routine \texttt{Proc} that is looking for \( x \) in a sorted list \( S \). The parameters are the ends of the list. While it is looking it does some processing in \texttt{ProcLow} and \texttt{ProcHigh}.

\texttt{Proc(lo,hi)}
\begin{itemize}
  \item If \( lo > hi \) we are done.
  \item \( k = \lfloor (lo + hi)/2 \rfloor \).
  \item If \( S[k] = x \), we are done.
  \item If \( S[k] < x \)
    \begin{itemize}
      \item Call \texttt{ProcHigh}(\( k,hi \)) and \texttt{Proc}(\( k + 1,hi \))
    \end{itemize}
  \item Else
    \begin{itemize}
      \item Call \texttt{ProcLow}(\( lo,k \)) and \texttt{Proc}(\( lo,k-1 \))
    \end{itemize}
\end{itemize}
End.

We begin by calling \texttt{Proc(1,\( n \))}. Most of the time is spent in \texttt{ProcLow} and \texttt{ProcHigh}. In fact, \texttt{ProcLow}(\( a,b \)) requires \( \lg(b - a + 1) \) basic operations and \texttt{ProcHigh}(\( a,b \)) requires \( (b - a + 1) \) basic operations. (You do \textit{not} need to know what any of this code is supposed to do.)

(a) Let \( W(n) \) be the worst case running time for \texttt{Proc(1,\( n \))}. Give a recursion and initial condition for \( W(2^n) \). (In the worst case, \( x \) is not in the list.)

\textbf{Ans.} When the length of the list is even, the part above \( k \) is exactly half of the list. The part below \( k \) is one shorter and also requires less processing time because of the “\lg”. Hence the worst case will be to always take the right half. Thus \( W(n) = W(n/2) + n/2 \). When \( n = 2^k \), \( W(2^k) = W(2^{k-1}) + 2^{k-1} \).

(b) Let \( A(n) \) be the average running time for \texttt{Proc(1, \( n \))}. Assuming \( x \) is not in the list and the probability that \( S[k] < x \) is \( 1/2 \), give a recursion for \( A(n) \). You need \textit{not} give an initial condition.

\textbf{Ans.} When \( n \) is even, the reasoning in the previous answer gives

\[
A(n) = \frac{A(n/2) + n/2}{2} + \frac{A(n/2 - 1) + \lg(n/2 - 1)}{2}.
\]

When \( n \) is odd, similar reasoning gives

\[
A(n) = \frac{A((n-1)/2) + (n-1)/2}{2} + \frac{A((n-1)/2) + \lg((n-1)/2)}{2}.
\]

There’s no need to write this as a single recursion, but you can. One way to do so is

\[
A(n) = \frac{A([((n-1)/2)]) + [(n-1)/2]}{2} + \frac{A([(n-1)/2)]) + \lg([(n-1)/2])}{2}.
\]
6. (65 pts.) Indicate whether true or false. Beware of guessing:

   correct answer +5pts.  incorrect answer −3pts.  no answer 0pts

T  $\Theta(2^{n+2}) = \Theta(2^n)$.  
T  $\Theta((n + 2)^2) = \Theta(n^2)$.  
F  $\Theta(2^{n+\log n}) = \Theta(2^n)$.  
T  $\Theta((n + \log n)^2) = \Theta(n^2)$.  

T Greedy algorithms are called “greedy” because they make the best choice at the present time, without concern for the future.

T Dynamic programming algorithms use a bottom up approach.

F Divide and conquer algorithms use a bottom up approach.

T If a divide and conquer algorithm requires recomputing the same quantity many times, it is a good idea to look for a dynamic programming algorithm.

T No greedy algorithm is known for the 0-1 Knapsack Problem.

F It is usually fairly easy to determine average and worst-case time complexities for backtracking algorithms.

F There is a search algorithm that uses comparison of keys and is significantly faster on average and in the worst case than binary search.

F There is a sorting algorithm that uses comparison of keys and is significantly faster on average and in the worst case than mergesort.

T Quicksort has a good average run time and a poor worst-case run time.