1. Work Problem 2.4 on page 71 of your textbook. In each case, let $n$ be the size of the original problem.

   - The time for algorithm A satisfies the recurrence $T(n) = 5T(n/2) + \Theta(n)$. The solution to this recurrence is $T(n) = \Theta(n \log_2 5)$.
   - The time for algorithm B satisfies the recurrence $T(n) = 2T(n-1) + \Theta(1)$. The solution to this recurrence is $T(n) = \Theta(2^n)$.
   - The time for algorithm C satisfies the recurrence $T(n) = 9T(n/3) + \Theta(n^2)$. The solution to this recurrence is $T(n) = \Theta(n^2 \log n)$, since $\log_3 9 = 2$.

   Algorithm C is fastest, since $\log_2 5 > 2$.

2. The following problem is similar to problem 3.5 on page 71 of your textbook. Solve the following recurrences. Give a $\Theta$ bound if possible; otherwise if an $\Omega$ or an $O$ bound.

   (a) $T(n) \leq 5T(n/5) + 1$.
      $T(n) = O(n)$.

   (b) $T(n) = 5T(n/5) + n$.
      $T(n) = \Theta(n \log n)$.

   (c) $T(n) \geq 5T(n/5) + n^2$. $T(n) = \Omega(n^2)$.

   (d) $T(n) = 25T(n/5) + n^2$. $T(n) = \Theta(n^2 \log n)$.

   (e) $T(n) \leq T(n-1) + n^4$.
      $T(n) = O(n^5)$.

   (f) $T(n) = 3T(n-1) + 1$.
      $T(n) = \Theta(3^n)$.

   (g) $T(n) \geq T(\sqrt{n}) + 1$. Let $T(n) = F(\ell)$ where $\ell = \log n$. (The base of the logarithm doesn’t matter.)
      $T(\sqrt{n}) = F(\log(\sqrt{n})) = F(\log n/2) = F(\ell/2)$. Rewriting the recurrence, we have $F(\ell) \geq F(\ell/2)+1$,
      hence $T(n) = F(\ell) = \Omega(\log \ell) = \Omega(\log \log n)$.

3. Work problem 2.13(a,b) in your textbook.

   (a) $B_3 = 1$, $B_5 = 2$, and $B_7 = 5$. 
$B_n = 0$ if $n$ is even, because a full binary tree must have an odd number of vertices.

(b) We have

\[
B_1 = 1, \\
B_3 = B_1B_1 = 1, \\
B_5 = B_3B_1 + B_1B_3 = 2, \\
B_7 = B_5B_1 + B_3B_3 + B_1B_5 = 5.
\]

In general $B_n = \sum_{i=1}^{n/2} B_{n-2i}B_{2i-1}$ for $n$ odd, $n \geq 3$.

The Catalan numbers are $1, 1, 2, 5, 14, 42, \ldots$ and $B_{2n+1}$ is the $n^{th}$ Catalan number

\[
C_n = \frac{1}{n+1} \binom{2n}{n}
\]


Let the array have indices starting with zero, as in C++. Our first phase sets $lo = 0$ and $hi$ to be some integer such that $A[hi] > x$. We initialize $hi = 1$, and keep doubling $i$ until $A[hi] > x$. Our loop invariant is that either $A[lo] = x$ for some $lo \leq i < hi$ or $x$ is not an entry of the array.

The second phase is to use binary search to halve the size of the search interval at each step, until $hi = lo + 1$. At this point, since the loop invariant still holds, either $A[lo] = x$ or $x$ is not in the array.

```cpp
int lo = 0;
int hi = 1;
while (A[hi] <= x) hi = 2*hi;
while (lo+1 < hi)
{
    int mid = (lo+hi)/2;
    if (A[mid] <= x) lo = mid; // loop invariant is maintained
    else hi = mid; // loop invariant is maintained
}
if (A[lo] == x)
    cout << "A[" << lo << "] = " << x << endl;
else
    cout << x << " is not an entry in the array" << endl;
```

Each phase takes $O(\log n)$ steps, hence the time complexity of the algorithm is $O(\log n)$.

5. Work problem 2.22 in your textbook.

Mr. Singh, I have deleted my work on this problem. The basic idea is not very difficult to understand, but the details are a killer. Hopefully, I can finish those details soon.
6. If \( f(n) \) is an increasing function, we say that \( f \) is polylogarithmic if \( \log(f(n)) = \Theta(\log \log n) \). We say that \( f \) is polynomial if \( \log(f(n)) = \Theta(\log n) \). We say that \( f \) is exponential if \( \log(f(n)) = \Theta(n) \).

It turns out that not every increasing function falls into one of those classes. Suppose \( F \) satisfies the recurrence:

\[
F(n) = F(n/2) + F(n-1) + 1
\]

It is obvious that \( n < F(n) < 2^n \), so \( F \) grows at least as fast as polynomial but no faster than exponential.

(a) Is \( F \) polylogarithmic? (Hint: No.)
(b) Is \( F \) polynomial? No. \( F \) grows faster than any polynomial function.
(c) Is \( F \) exponential? No. \( F \) grows slower than any exponential function.

Not every polynomial function (as defined above) is \( \Theta(n^K) \) for some constant \( K > 1 \), and not every exponential function (as defined above) is \( \Theta(2^{Cn}) \) for some constant \( C > 0 \). However, these simplifications are “almost” true: more specifically, every polynomial function is both \( O(n^{K_1}) \) and \( \Omega(n^{K_2}) \) for constants \( K_1 \geq K_2 > 1 \), while every exponential function is both \( O(2^{C_1n}) \) and \( \Omega(2^{C_2n}) \) for constants \( C_1 \geq C_2 > 0 \).

We first note that \( F(n) \geq F(n-1) + 1 \), hence \( F \) is monotone increasing. The formula that defines \( F \) works when you substitute any quantity for \( n \), such as \( n-1, n-2 \), etc. Thus

\[
\begin{align*}
F(n) &= F(n/2) + F(n-1) + 1 \\
F(n-1) &= F((n-1)/2) + F(n-2) + 1 \\
F(n-2) &= F((n-2)/2) + F(n-3) + 1 \\
\text{and so forth. Substituting, we obtain} \\
F(n) &= F(n/2) + F((n-1)/2) + F((n-2)/2) + F(n-3) + 3 \\
\text{Repeated substitution yields, for any } m \\
F(n) &= F(n/2) + F((n-1)/2) + \cdots + F((n-m+1)/2) + F(n-m) + m \\
\text{Let } m = n/2 \text{ (assuming that is an integer)} \\
F(n) &= F(n/2) + F((n-1)/2) + \cdots + F((n/2+1)/2) + F(n/2) + n/2 \\
\text{Since } F \text{ is monotone increasing, we have two inequalities} \\
F(n) &\geq nF(n/4)/2 + F(n/2) + n/2 \\
F(n) &\leq nF(n/2)/2 + F(n/2) + n/2
\end{align*}
\]

To make the problem easier to understand, we make the simplifying assumption that a polynomial function of \( n \) is of the form \( n^K \) for some constant \( K \), and that an exponential function of \( n \) is of the form \( 2^{Cn} \) for some constant \( C \).

Suppose \( F(n) = n^K \) for some constant \( K \). Then

\[
\begin{align*}
F(n) &\geq nF(n/4) \\
n^K &\geq \frac{n^{K+1}}{4^K} \\
4^K &\geq n
\end{align*}
\]
Constradiction, since $4^K$ is constant and $n$ is arbitrarily large.

Suppose $F(n) = 2^{Cn}$ for some constant $C$. Then

\[
2^{Cn} \leq 2^{Cn/2} n + 2^{Cn/2} + n/2
\]

divide both sides by $2^{Cn/2}$

\[
2^{Cn/2} \leq (n + 1) + \frac{n}{2^{Cn/2 - 1}}
\]

Contradiction, since an exponential function grows faster than a polynomial function.