Final Exam
Solution Manual

Problem 1

1. True.

   If NPC = P, every NP problem can be reduced to a NPC problem and solved in polynomial time, which implies NP = P.

2. False.

   We do not know whether L_2 in NP or not, thus we can not say $L_2 \in$ NPC.

3. False.

   The complexity class NP represents the problems which "can be verified" within polynomial time.

4. False.

   When performing an amortized analysis, we usually require the total amortized cost to be a "upper bound" of the total actual cost, because we do not want to underestimate the cost of an algorithm.

Problem 2

1. As follows.

$$T_1(A \cup B \cup C \cup D) = T_1(A) + T_1(B) + T_1(C) + T_1(D)$$
$$T_\infty(A \cup B \cup C \cup D) = \max(T_\infty(A) + \max(T_\infty(B), T_\infty(C)), T_\infty(D))$$

2. Because they need to get up to speed and will probably be working at low efficiency for several months and dragging down the efficiency of the people who have to mentor them.

   We should find some features to delete by their importance, and maybe ship them in the future updated version.

3. See `http://www.joelonsoftware.com/items/2007/10/26.html`.

4. As follows.

```
NEW_RACE_EXAMPLE(n)
    fractorial = 1
    parallel for i = 1 to n
        fractorial *= i
    print fractorial
```

5. Here we are using the simple triangle inequality $|A + B| \leq |A| + |B|$.

   Let the load factor just after i-th deletion be α_i, we can divide it into 2 cases.

   (1) If $\alpha_i \geq \frac{1}{3}$, which means we would NOT contract the table, we have

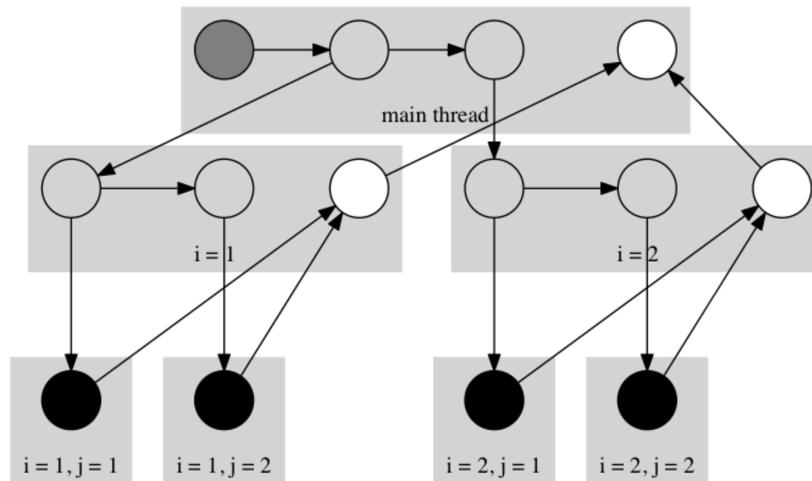
$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + |2 \cdot T.\text{num}_i - T.\text{size}_i| - |2 \cdot T.\text{num}_{i-1} - T.\text{size}_{i-1}| \\
&= 1 + |2 \cdot (T.\text{num}_{i-1} - 1) - T.\text{size}_{i-1}| - |2 \cdot T.\text{num}_{i-1} - T.\text{size}_{i-1}| \\
&= 1 + |2 \cdot T.\text{num}_{i-1} - T.\text{size}_{i-1} - 2| - |2 \cdot T.\text{num}_{i-1} - T.\text{size}_{i-1}| \\
&\leq 1 + |2 \cdot T.\text{num}_{i-1} - T.\text{size}_{i-1}| + 2 - |2 \cdot T.\text{num}_{i-1} - T.\text{size}_{i-1}| \\
&= 3
\end{aligned}$$

   (2) If $\alpha_i < \frac{1}{3}$, which means we would contract the table. When T.size sufficiently large, it implies $\alpha_{i-1} < \frac{1}{2}$, thus we have

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + T.\text{num}_i + |2 \cdot T.\text{num}_i - T.\text{size}_i| - |2 \cdot T.\text{num}_{i-1} - T.\text{size}_{i-1}| \\
&= 1 + T.\text{num}_i + \left|2 \cdot (T.\text{num}_{i-1} - 1) - \frac{2}{3} \cdot T.\text{size}_{i-1}\right| - |2 \cdot T.\text{num}_{i-1} - T.\text{size}_{i-1}| \\
&= 1 + T.\text{num}_i + \left|2 \cdot T.\text{num}_{i-1} - \frac{2}{3} \cdot T.\text{size}_{i-1} - 2\right| - |2 \cdot T.\text{num}_{i-1} - T.\text{size}_{i-1}| \\
&\leq 3 + T.\text{num}_i + \left|2 \cdot T.\text{num}_{i-1} - \frac{2}{3} \cdot T.\text{size}_{i-1}\right| - |2 \cdot T.\text{num}_{i-1} - T.\text{size}_{i-1}| \\
&= 3 + T.\text{num}_i + (2 \cdot T.\text{num}_{i-1} - \frac{2}{3} \cdot T.\text{size}_{i-1}) + (2 \cdot T.\text{num}_{i-1} - T.\text{size}_{i-1}) \\
&\leq 3 + T.\text{num}_i + 2 \cdot \left(\frac{1}{3} \cdot T.\text{size}_{i-1} + 1\right) - \frac{2}{3} \cdot T.\text{size}_{i-1} + 2 \cdot \left(\frac{1}{3} \cdot T.\text{size}_{i-1} + 1\right) - T.\text{size}_{i-1} \\
&\leq 5 + T.\text{num}_i + 2 \cdot \left(\frac{1}{3} \cdot T.\text{size}_{i-1} + 1\right) - T.\text{size}_{i-1} \\
&\leq 7 + T.\text{num}_i - \frac{1}{3} \cdot T.\text{size}_{i-1} \\
&\leq 7
\end{aligned}$$

Problem 3

1. As follows.



   The black nodes represent strands corresponding to the procedure that calculates `c[i][j]`.

   The white nodes represent strands corresponding to the procedure that returns.

   The grey node in main thread represent the strand corresponding to the first to lines in this function.

   The remaining nodes represent strands corresponding to the procedure that implements parallel for.

2. work: $\Theta(n^3)$, which is simply derived from the 3-level nested for loops.

   span: $\Theta(\lg n) + \Theta(\lg n) + \Theta(n) = \Theta(n)$, which is simply derived from "parallel for", "parallel for", and "for" loops.

   parallelism: work/span $= O(n^2)$.

3. As follows. Note that we are using 1-base indexed array here.

```
1  Inner-Product(x, y, l, r)
2      if ( l == r ) retrun x[l] * y[l]
3      m = (l + r) / 2
4      sl = spawn Inner-Product(x, y, l, m)
5      sr = spawn Inner-Product(x, y, m + 1, r)
6      sync
7      return sl + sr
8
9  P-Square-Matrix-Multiply(a, b)
10     n = a.rows
11     let c be a new n x n matrix
12     let x be a new n-vector
13     let y be a new n-vector
14     parallel for i = 1 to n
15         parallel for j = 1 to n
16             parallel for k = 1 to n
17                 x[k] = a[i][k]
18                 y[k] = b[k][j]
19             c[i][j] = Inner-Product(x, y, 1, n)
20     return c
```

4. It is easy to see that the span of algorithm "Inner-Product" equals to the depth of recursion, which is $T(n) = T(n/2) + O(1) = O(\lg n)$. Thus the total span is $O(\lg n) + O(\lg n) + O(\lg n) = O(\lg n)$.

Problem 4

1. Can we find a schedule for a subset of the given tasks or all the tasks so that the amount of profit obtained is no less than a given number P?

2. It is easy to see that there is a polynomial time algorithm which can verify the decision problem, which simply make the scheduling as the certificate, and check whether this scheduling is valid and can obtain at least P profit, so this decision problem is in NP.

   We can reduce the decision problem version of 0-1 knapsack problem to this problem, which is a well-known NPC problem, so that every NP problem can be reduced to this problem, with the above statement which make we knew that this problem is in NP, this problem is in NPC.

   The reduction is shown as follows. We can make n tasks as the number of items, and $p_i = v_i$, $t_i = w_i$, $s_i = 0$, $D = W$.

3. Without loss of generality, we can assume that s_i is sorted in non-decreasing order.

   Suppose that in the optimal solution of scheduling we execute the k-th task last, and start from d-th unit time. The way we schedule the first $k-1$ tasks within first d units time must optimal, since if there is way with more profit, we can simply copy-and-paste that way into the original solution, and produce another solution whose profit is more than optimum, a contradiction.

4. As follows.

```
SOLVE(N, D, P, t, p, s)
    Sort tasks by ready time in non-decreasing time
    D = min(D, N + N * N)
    dp = new array[N+1][D+1] with zero
    for i = 1 to N
        for j = 0 to D
            dp[i][j] = dp[i-1][j]
            if ( j >= s[i] + t[i] )
                dp[i][j] = max(dp[i][j], dp[i-1][j-t[i]] + p[i])
    return dp[N][D] >= P
```

   Note that we can bound D by $N + N^2$ under the constraint stated in this problem, which is the upper bound of the last finishing task time. So this algorithm runs in polynomial time.

Problem 5

1. First, we do a in-order traversal in the subtree rooted at x as follows, which store a sorted list of node in $O(x.\text{size})$ time and space.

```
list = []
IN-ORDER(x)
    if ( x == NULL ) return
    IN-ORDER(x.left)
    list.append(x)
    IN-ORDER(x.right)
```

   Second, we run a recursive build algorithm on this list as follows.

```
BUILD(list)
    if ( list.empty() ) return NULL
    mid = list.size / 2
    list[mid].left  = BUILD(list[:mid])
    list[mid].right = BUILD(list[mid+1:])
    return list[mid]
```

   The "BUILD" algorithm also runs in $T(N) = 2T(N/2) + O(1) = O(N) = O(x.\text{size})$.

2. In fact, it takes $O(\log_{1/\alpha} n)$, which is $O(\log n)$ when α is a given constant. The worse-case time need to perform a search in a binary search tree is the height of tree, which is defined recursively, thus we can show that by induction on n. When $n = 1$, $\log_{1/\alpha} n = 0$ is good for us. When $n > 1$, assume that the claim is correct for all $n' < n$, we have

$$\begin{aligned}
\text{height}(x) &= 1 + \max(\text{height}(x.\text{left}), \text{height}(x.\text{right})) \\
&\leq 1 + \log_{1/\alpha} \max(x.\text{left.size}, x.\text{right.size}) \\
&\leq 1 + \log_{1/\alpha} \alpha n \\
&= \log_{1/\alpha} 1/\alpha + \log_{1/\alpha} \alpha n \\
&= \log_{1/\alpha} n
\end{aligned}$$

3. By definition, we know $\Delta(x)$ is an absolute value, which is non-negative, and the summation over non-negative value times a non-negative constant c will all be non-negative, thus any binary search tree has non-negative potential.

   Without loss of generality we assume that $x.\text{left.size} > x.\text{right.size}$. If $\Delta(x) \geq 2$, we have

$$\begin{aligned}
x.\text{left.size} &\geq x.\text{right.size} + 2 \\
&= x.\text{size} - 1 - x.\text{left.size} + 2 \\
&= x.\text{size} - x.\text{left.size} + 1
\end{aligned}$$

   , which implies $x.\text{left.size} > \frac{1}{2} x.\text{size}$.

   Thus for any (1/2)-balanced binary search tree T, any node x of T must satisfy $\Delta(x) < 2$, which make $\Phi(T) = 0$.

4. Suppose the m-node subtree we are rebuilding is rooted at x, since it is not α-balanced, without loss of generality we can assume that $x.\text{left.size} > \alpha m$, and $x.\text{right.size} < (1 - \alpha)m$.

   Consider the potential of that subtree before the rebuilding, we have $\Phi(T) \geq c\Delta(x) \geq c(2\alpha - 1)m$, and the potential of that subtree after the rebuilding will be zero, since it is (1/2)-balanced. Thus the change of potential will be at least $c(2\alpha - 1)m$.

   By letting $c \geq 1/(2\alpha-1)$, the potential will drop by at least m when rebuilding, which makes the amortized time to rebuild be $O(1)$.

   Note that this bound is tight, since we can easily make the change of potential approach to $c(2\alpha - 1)m$ by constructing a simple tree rooted at x, while x.left and x.right are (1/2)-balanced with proper size.

5. The insertion of deletion without rebuilding takes $O(\text{height})$ time, which is $O(\log n)$ as argued in 2.

   Note that both operation will only effect the balancing state of one path on tree, thus the change of potential will be $O(\log n)$, and we will need to rebuild at most one subtree after each operation, which is $O(1)$ as argued in 4.

Problem 6

Meow(?)