7. What is the asymptotical running-time complexity of Find-Array-Max?

```
Function FIND-ARRAY-MAX(A, n)
1: if (n = 1) then
2:     return(A[1])
3: else
4:     return(max(A[n], FIND-ARRAY-MAX (A, n − 1)))
5: end if
```

**Solution:**

**T(n) = T(n-1) + 1, T(1) = 0.**

**T(n) = Θ(n)**

8. Mr. Smith has an algorithm which he has proved (correctly) to run in time O(2ⁿ). He coded the algorithm correctly in C, yet he was surprised when it ran quickly on inputs of size up to a million. What are at least two plausible explanations of this behavior?

**Solution:**

- **Perhaps the algorithm runs in linear time. O(2ⁿ) means at most time c2ⁿ. Even a linear algorithm is O(2ⁿ).**
- **Perhaps the input to the program is not the worst case input. It could be the "easiest" input. The exponential behavior doesn't "kick in" until n is huge.**

9. Given a set $\{x_1 \le x_2 \le \ldots \le x_n\}$ of points on the real line, determine the smallest set of unit-length closed intervals (e.g. the interval [1.25,2.25] includes all $x_i$ such that $\{1.25 \le x_i \le 2.25\}$ ) that contains all of the points. Give the most efficient algorithm you can to solve this problem, prove it is correct and analyze the time complexity.

The greedy algorithm we use is to place the first interval at $[x_1, x_1 + 1]$, remove all points in $[x_1, x_1 + 1]$ and then repeat this process on the remaining points.

Clearly the above is an $O(n)$ algorithm. We now prove it is correct.

**Greedy Choice Property:** Let $S$ be an optimal solution. Suppose $S$ places its leftmost interval at $[x, x + 1]$. By definition of our greedy choice $x \le x_1$ since it puts the first point as far right as possible while still covering $x_1$. Let $S'$ be the scheduled obtained by starting with $S$ and replacing $[x, x + 1]$ by $[x_1, x_1 + 1]$. We now argue that all points contained in $[x, x + 1]$ are covered by $[x_1, x_1 + 1]$. The region covered by $[x, x + 1]$ which is not covered by $[x_1, x_1 + 1]$ is $[x, x_1)$ which is the points from $x$ up until $x_1$ (but not including $x_1$). However, since $x_1$ is the leftmost point there are no points in this region. (There could be additional points covered by $[x + 1, x_1 + 1]$ that are not covered in $[x, x + 1]$ but that does not affect the validity of $S'$). Hence $S'$ is a valid solution with the same number of points as $S$ and hence $S'$ is an optimal solution.

**Optimal Substructure Property:** Let $P$ be the original problem with an optimal solution $S$. After including the interval $[x_1, x_1 + 1]$, the subproblem $P'$ is to find an solution for covering the points to the right of $x_1 + 1$. Let $S'$ be an optimal solution to $P'$. Since, $\text{cost}(S) = \text{cost}(S') + 1$, clearly an optimal solution to $P$ includes within it an optimal solution to $P'$.

10. You are going on another long trip (this time your headlights are working). You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \ldots < a_n$, where each $a_i$ is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance an), which is your destination.

You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the penalty for that day is $(200\text{-}x)^2$. You want to plan your trip so as to minimize the total penalty – that is the sum, over all travel days, of daily penalties. Give an efficient algorithm that determines the minimum penalty for the optimal sequence of hotels at which to stop.

**Let S[j] be the minimum total penalty when you stop at hotel j.**

**Let S[0] = 0**
**For j >= 1, j <= n**
**S[j] = Inf**
**For i = 0, i < j**
**S[j] = min { S[j], S[i] + (200- $(a_j - a_i)$)² }**
**Return: S[n]**

11. ( True / False )

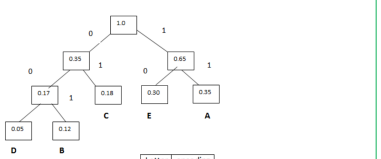
a ) $n^2 + n = O(n^2)$? **- true**

b) $\lg(n^2) = O(n)$? **true**

c) A function that calls a O(n) function three times and has constant time for the rest of the algorithm. The overall asymptotic run-time of the algorithm is O(n). **True**

12. Suppose we have an alphabet with only five letters A, B, C, D, E which occur with the following frequencies. Construct a Huffman code.

| Letter | A | B | C | D | E |
|---|---|---|---|---|---|
| frequency | 0.35 | 0.12 | 0.18 | 0.05 | 0.30 |




| Letter | encoding |
|---|---|
| A | 11 |
| B | 001 |
| C | 01 |
| D | 000 |
| E | 10 |

13. For each of the following give a tight Θ( ) bound on the number of times the $z \leftarrow z + 1$ statement is executed and justify your solution.

```
j ← 0
while (j < n) do
    j ← j + 2
    z ← z + 1
```

AN ANSWER. Since $j$ goes through the values 0, 2, 4, 6, ... until $j$ reaches $n$ (if $n$ is even) or $n+1$ (if $n$ is odd), the while loop goes through at most $\lceil n/2 \rceil$ many iterations. Hence, $z$ is in $\Theta(n)$.

```
for k ← 0 to n do
    for j ← 0 to k do
        z ← z + 1
```

AN ANSWER. *Inner loop:* Since $j$ goes from 0 to $k$, the inner loop has $(k + 1)$-many iterations. So, $z$ is increased by $(k + 1)$. *Outer loop:* $k$ goes from 0 to $n$. So $z$ is increased by
$\sum_{k=0}^{n}(k+1)$
$= 1 + 2 + \cdots + n + (n+1)$
$= \frac{(n+1)(n+2)}{2} \in \Theta(n^2)$.

```
i ← n
while (i > 1) do
    i ← ⌊i/2⌋
    z ← z + 1
```

AN ANSWER. Since $i$ takes on the sequence of values $n = n/2^0$, $\lceil n/2 \rceil = \lceil n/2^1 \rceil$, $\lceil n/4 \rceil = \lceil n/2^2 \rceil, \ldots, \lceil n/2 \rceil$ until $i$ is $\le 1$. The smallest value of $i$ such that $n/2^i \le 1$ is $\lceil \log_2 n \rceil$. So there are $(1 + \lceil \log_2 n \rceil)$-many iterations and $z \in \Theta(\log_2 n)$.

14. You just started a consulting business where you collect a fee for completing various types of projects (the fee is different for each project). You can select in advance the projects you will work on during some finite time period. You work on only one project at a time and once you start a project it must be completed to receive your fee. There is a set of n projects $p_1, p_2, \ldots p_n$ each with a duration $d_1, d_2, \ldots d_n$ (in days) and you receive the fee $f_1, f_2, \ldots, f_n$ (in dollars) associated with it. That is project $p_i$ takes $d_i$ days and you collect $f_i$ dollars after it is completed.

Each of the n projects must be completed in the next D days or you lose its contract. Unfortunately, you do not have enough time to complete all the projects. Your goal is to select a subset S of the projects to complete that will maximize the total fees you earn in D days.

(a) What type of algorithm would you use to solve this problem? Divide and Conquer, Greedy or **Dynamic Programming.** Why? **It is similar to the 0-1 knapsack.**

(b) Describe the algorithm verbally. If you select a DP algorithm give the formula used to fill the table or array.

**Let OPT(i,d) be the maximum fee collected for considering projects 1,..., i with d days available.**

**The base cases are OPT(i, 0) = 0 for i = 1, ..., n and OPT(0, d) = 0 for d = 1, ..., D.**

```
For i = 1 to n {
  For j = 1 to D {          //updated from d to j
      If (di > j ) {
          OPT(i, j) = OPT( i-1, j)   // not enough time to complete project i }
      Else {
          OPT(i, j) = max ( OPT( i-1, j),   // Don't complete project i
                            OPT( i-1, j-di) + fi  ) //  Complete project I and earn fee fi
                          )
      }
  }
}
```

(c) What is the running time of your algorithm?

**O(nD) or Θ(nD)**

Define $C[i]$ in terms of earlier table entries. Indeed its clear that $C[i] = C[k] + R[k,i]$. Since we do not know the post $k$ beforehand, we take the minimum of this expression over all $k$ in the range $1 \le k < i$. Define

$$C[i] = \begin{cases} 0 & i = 1 \\ \min_{1 \le k < i}(C[k] + R[k,i]) & 1 < i \le n \end{cases}$$

With this formula, the algorithm for filling in the table is straightforward.

```
CanoeCost(R)
1.  n ← #rows[R]
2.  C[1] ← 0
3.  for i ← 2 to n
4.      min ← R[1,i]
5.      for k ← 2 to i − 1
6.          if C[k] + R[k,i] < min
7.              min ← C[k] + R[k,i]
8.      C[i] ← min
9.  return C[n]
```

To determining the actual sequence of canoe rentals which minimizes cost alter the CanoeCost() algorithm so as to construct the optimal sequence while the table $C[1...n]$ is being filled. In the following algorithm we maintain an array $P[1...n]$ where $P[i]$ is defined to be the post $k$ at which the last canoe is rented in an optimal sequence from 1 to $i$. Note that the definition of $P[1]$ can be arbitrary since it is never used. Array $P$ is then used to recursively print out the sequence.

```
CanoeSequence(R)
1.  n ← #rows[R]
2.  C[1] ← 0 , P[1] ← 0
3.  for i ← 2 to n
4.      min ← R[1,i]
5.      P[i] ← 1
6.      for k ← 2 to i − 1
7.          if C[k] + R[k,i] < min
8.              min ← C[k] + R[k,i]
9.              P[i] ← k
10.     C[i] ← min
11. return P
```

```
PrintSequence(P, i)   (Pre: 1 ≤ i ≤ length[P])
1.  if i > 1
2.      PrintSequence(P, P[i])
3.      print "Rent a canoe at post " P[i] " and drop it off at post " i
```

Both CanoeCost() and CanoeSequence() run in time $\Theta(n^2)$, since the inner for loop performs $i - 2$ comparisons in order to determine $C[i]$, and $\sum_{i=2}^{n}(i-2) = \frac{(n-1)(n-2)}{2} = \Theta(n^2)$. The top level call to PrintSequence(*P, n*) has a cost that is the depth of the recursion, which is in turn, the number of canoes rented in the optimal sequence from post 1 to post *n*. Thus in worst case, PrintSequence() runs in time $\Theta(n)$ .

15. Product Sum

Given a list of n integers, $v_1, \ldots, v_n$, the product-sum is the largest sum that can be formed by multiplying adjacent elements in the list. Each element can be matched with at most one of its neighbors.

For example, given the list 1, 2, 3, 1 the product sum is 8 = 1 + (2 × 3) + 1, and given the list 2, 2, 1, 3, 2, 1, 2, 2, 1, 2 the product sum is 19 = (2 × 2) + 1 + (3 × 2) + 1 + (2 × 2) + 1 + 2.

**Solutions:**

**a) 1 + (4x3) + 2 + (3x4) + 2 = 29**

**b) OPT[j] =** $\begin{cases} 0 & \text{if } j = 0 \\ v_1 & \text{if } j = 1 \\ \max\{\text{OPT}[j-1] + v_j, \text{OPT}[j-2] + v_j * v_{j-1}\} \end{cases}$

**c) running time Θ(n)**

## The Subset-Sum problem

- Input: An array A[1 . . . n] of positive integers, and a target integer T > 0.
- Output: True iff there is a subset of the A values that sums to T .
- Example
  - Input: A = [1, 3, 4, 6, 10 ] and T = 16
  - Output: True
- If we look at all subsets for a sum of T then the running time is exponential.

## The Subset-Sum problem

Outline:

Let S[i, j] be defined as true iff there is a subset of elements A[1 . . . i] that sums to j .

Then S[n, T] is the solution to our problem.

In general:

S[i, j] = S[i − 1, j − A[i]] ∨ S[i − 1, j]

With initial conditions

S[i, 0] = True, and S[0, j] = False, for j > 0.

## Translating to pseudo-code

```
for (i = 0; i <= n; i ++)
    S [i, 0]= TRUE;
for (j = 1; j <= T; j ++)
    S [0,j] = FALSE;
for (i = 1; i <= n; i ++)
    for (j = 1; j <= T; j ++)
        if j < A[i] then
            S [i,j] = S [i -1, j];
        else
            S [i,j] = S [i -1, j] || S [i-1, j-A[i]]);
```

## Time complexity is Θ(nT).

| | $T_1(n)$ | $T_2(n)$ | Is $T_1(n) = O(T_2(n))$? | Is $T_1(n) = \Omega(T_2(n))$? | Is $T_1(n) = \Theta(T_2(n))$? |
|---|---|---|---|---|---|
| a | $25n \ln n + 5n$ | $\frac{1}{2} n \log_2 n$ | yes | yes | yes |
| b | $\frac{1}{2} n^2 + n \log_2 n$ | $5n \log_2 n$ | no | yes | no |
| c | $\sqrt{n}(\log_2 n)$ | $n$ | yes | no | no |
| d | $2^{\log_2 n}$ | $2n^2$ | yes | no | no |
| e | $n\sqrt{n}$ | $n^{1.4}$ | no | yes | no |

**Algorithm *prefixAverages1* (*X, n*)**

| **Input** array $X$ of $n$ integers | |
|---|---|
| **Output** array $A$ of prefix averages of $X$ | #operations |
| $A \leftarrow$ new array of $n$ integers | $n$ |
| **for** $i \leftarrow 0$ **to** $n - 1$ **do** | $n$ |
| $s \leftarrow X[0]$ | $n$ |
| **for** $j \leftarrow 1$ **to** $i$ **do** | $1 + 2 + \ldots + (n-1)$ |
| $s \leftarrow s + X[j]$ | |
| $A[i] \leftarrow s / (i + 1)$ | $n$ |
| **return** $A$ | 1 |

**Algorithm *prefixAverages2* (*X, n*)**

| **Input** array $X$ of $n$ integers | |
|---|---|
| **Output** array $A$ of prefix averages of $X$ | #operations |
| $A \leftarrow$ new array of $n$ integers | $n$ |
| $s \leftarrow 0$ | 1 |
| **for** $i \leftarrow 0$ **to** $n - 1$ **do** | $n$ |
| $s \leftarrow s + X[i]$ | $n$ |
| $A[i] \leftarrow s / (i + 1)$ | $n$ |
| **return** $A$ | 1 |

## Algorithm *prefixAverages2* runs in $\Theta(n)$ time

**Observation**: We can assume that all jobs finish after $n$ minutes. Suppose not. So there is an empty minute starting at say $0 \le k \le n - 1$ (where no job is scheduled) and there is a job $j_i$ for some $1 \le i \le n$ which is scheduled some time after n minutes. If we schedule job $j_i$ to start at minute $k$, then this can only be a better solution since everything remains same except $j_i$ might now be able to meet its deadline.


We assign time intervals $M_i$ for $1 \le i \le n$ where $M_i$ starts at minute $i - 1$ and ends at minute $i$. The greedy algorithm is as follows:

- Arrange the jobs in decreasing order of the penalties $p_1 \ge p_2 \ge \cdots \ge p_n$ and add them in this order
- To add job $j_i$, if any time interval $M_l$ is available for $1 \le l \le d_i$, then schedule $j_i$ in the last such available interval. Else schedule $j_i$ in the first available interval starting backwards from $M_n$

Let S be the greedy schedule and suppose it schedules job $j_i$ at time $t_i$ for each $1 \le i \le n$. We show by induction that there is an optimal schedule $T_r$ that agrees with S on the schedule of the first $r$ intervals for $1 \le r \le n$.

**Base Case:** The base case is when $r = 1$. Let S' be an optimal schedule. Suppose $S'$ schedules $j_1$ at time $t \ne t_1$. By our observation above, $S'$ must schedule some job $j_k$ at time $t_1$. Let $T_1$ be the schedule obtained by swapping times of $j_1$ and $j_k$ in S'. Note that $T_1$ makes the greedy choice to schedule $j_1$. In each of the following 3 cases we show that penalty of $T_1$ is at most that of S'.

- $d_1 \ge t_1$ and $j_1$ does not incur penalty in S'. This implies $t_1 > t$ since $t_1$ was the last available slot for $j_1$ in greedy algorithm. So in $T_1$, $j_1$ still does not incur penalty and $j_k$ gets scheduled at $t$ which is earlier than $t_1$.
- $d_1 \ge t_1$ and $j_1$ incurs penalty in S'. This implies $t > d_1 \ge t_1$. In $T_1$ we do not pay penalty for $j_1$ since $d_1 \ge t_1$. Worst case we might pay the penalty $p_k$ for $j_k$. So the net difference is penalty is $p_k - p_1 \le 0$.
- $d_1 < t_1$. Hence there is no way to schedule $j_1$ without incurring penalty. Hence $j_1$ pays $p_1$ in both S' and $T_1$. In greedy algorithm since $t_1$ is chosen as last available slot, we have $t_1 > t$. In T we schedule $j_k$ at t instead of $t_1$, which can only reduce the penalty of $j_k$.

**Inductive Hypothesis**: There is an optimal schedule $T_r$ that agrees with S on the schedule of the first $r$ intervals.

**Inductive Step**: If $r = n - 1$, then by the Observation above the job $j_n$ must be scheduled in the only remaining slot from 1 to n. If $r < n - 1$ then consider the optimal schedule $T_r$. Let it schedule the job $j_{r+1}$ at time $t \ne t_{r+1}$. By Observation above, $T_r$ schedules a job say $j_h$ at time $t_{r+1}$. Since $T_r$ agrees with S on first $r$ jobs we have $h > r + 1$. Let $T_{r+1}$ be the schedule that swaps times of $j_h$ and $j_{r+1}$, *i.e.*, schedules $j_h$ at time $t$ and $j_{r+1}$ at time $t_{r+1}$. Note that $T_{r+1}$ agrees with S on first $r + 1$ jobs. A very similar argument as in the base case shows that penalty of $T_{r+1}$ is at most as much as the penalty of $T_r$. Then the optimality of $T_r$ implies the optimality of $T_{r+1}$ as well.