Quiz 1

- In Class paper quiz
- Closed book quiz
- Note card 3”x5”
- Calculator allowed
- T/F, multiple choice, essay, code

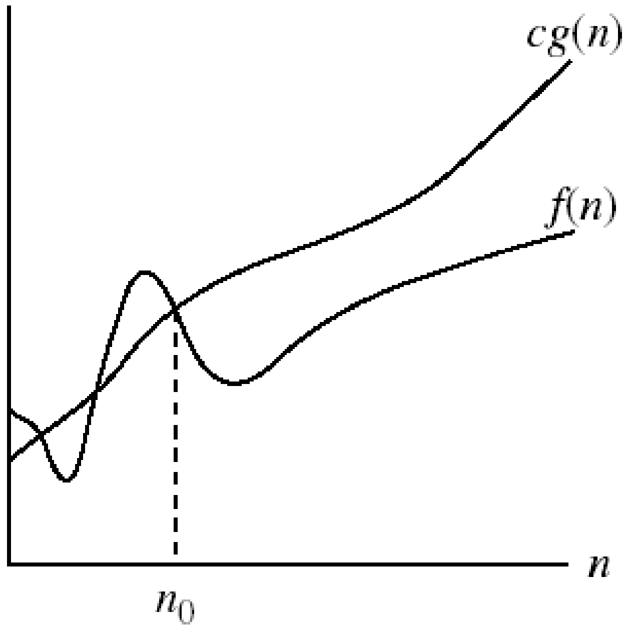
Topics covered

- Asymptotic Growth
  - Compare order of growth
  - Prove asymptotic notation using basic definition
  - Worst case and average case analysis
- Analyzing non-recursive algorithms
  - Sum of arithmetic series
  - Sum of geometric series
- Analyzing recursive algorithms
  - Defining recurrence
  - Solving recurrence
    - Master theorem
    - Iteration
    - Recursion tree method
    - Substitution method
- Divide and Conquer algorithms
- Dynamic Programming

Asymptotic notations

- O-notation

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}\ .$$

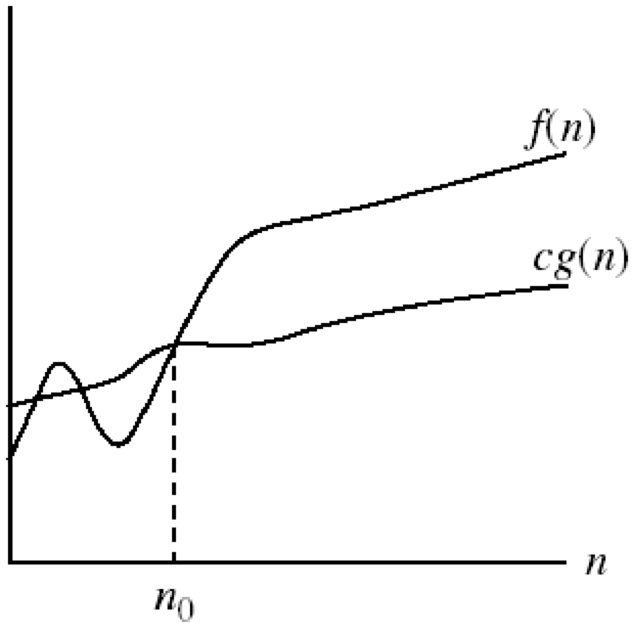


$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Asymptotic notations (cont.)

- Ω - notation

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



ns

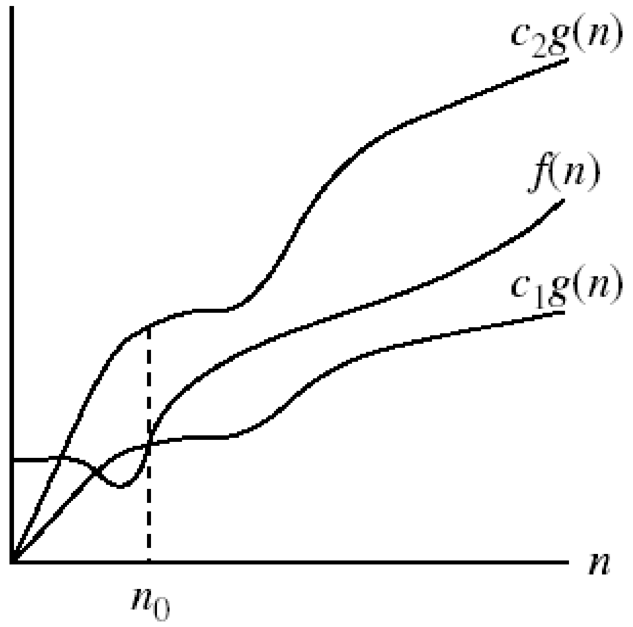
of

$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Asymptotic notations (cont.)

- Θ-notation

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$



ne set of functions

me order of growth

t or equivalence class

$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Prove or give counter example

If $f_1(n)=O(g_1(n))$ and $f_2(n)=O(g_2(n))$ then $f_1(n)*f_2(n) = O(g_1(n)g_2(n))$

Limit Method: The Process

- Say we have functions f(n) and g(n). We set up a limit quotient between f and g as follows

$$\lim_{n\to\infty} f(n)/g(n) = \begin{cases} 0 & \text{Then } f(n)\in O(g(n)) \\ c>0 & \text{Then } f(n)\in \Theta(g(n)) \\ \infty & \text{Then } f(n)\in \Omega(g(n)) \end{cases}$$

- The above can be proven using calculus, but for our purposes, the limit method is sufficient for showing asymptotic inclusions
- Always try to look for algebraic simplifications first
- If f and g both diverge or converge on zero or infinity, then you need to apply the l'Hôpital Rule

	$T_1(n)$	$T_2(n)$	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	$25n \ln n + 5n$	$\frac{1}{2} n \log_2 n$			
b	$\frac{1}{2} n^2 + n \log_2 n$	$5n \log_2 n$			
c	$\sqrt{n}(\log_2 n)$	n			
d	$2^{\log_2 n}$	$2n^2$			
e	$n\sqrt{n}$	$n^{1.4}$			

	$T_1(n)$	$T_2(n)$	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	$\frac{1}{2} n^2 + n \log_2 n$	$5n \log_2 n$			
c	$\sqrt{n}(\log_2 n)$	n			
d	$2^{\log_2 n}$	$2n^2$			
e	$n\sqrt{n}$	$n^{1.4}$			

	$T_1(n)$	$T_2(n)$	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			
d	$2^{\log_2 n}$	$2n^2$			
e	$n\sqrt{n}$	$n^{1.4}$			

	$T_1(n)$	$T_2(n)$	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	$n\sqrt{n}$	$n^{1.4}$			

	$T_1(n)$	$T_2(n)$	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

5. Order the following functions in increasing order of asymptotic (big-O) complexity.

Solution:

f(n), p(n), g(n) q(n), h(n)

Theoretical Running times

- Iterative algorithm
  - Translate loops in code to summations
  - Find closed form of summation (solve)

- Recursive Algorithm
  - Translate recursion into a recurrence
  - Find a closed form for recurrence (solve)

Find running time

```
Foo(n) {
    total = 0
    if n = 1 return 2
    else {
        total = Foo(n/2) + Foo(n/2)
        for i = 1 to n do
            for j = 1 to n do
                for k = 1 to n do
                    total = total + k
        return total  }
}
```

Recurrence

$$T(n) = 2T(n/2) + n^3$$

Master Method

$$T(n) = 2T(n/2) + n^3$$

Case 3

$$\left.\begin{aligned} &a = 2, b = 2, f(n) = n^3 \\ &n^{\log_2 2} = n = \Theta(n) \\ &f(n) = n^3 = \Omega(n^{\log_2 2+\varepsilon}), \varepsilon = 2 \end{aligned}\right\}$$

$$af(n/b) \le cf(n), c < 1$$

$$2(n/2)^3 = 1/4\, n^3 \le cn^3, c = 1/4$$

$$T(n) = \Theta(n^3)$$

Master's method

- "Cookbook" for solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, $a \geq 1$, $b > 1$, and $f(n) > 0$

Case 1: if $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then: $T(n) = \Theta(n^{\log_b a})$

Case 2: if $f(n) = \Theta(n^{\log_b a})$, then: $T(n) = \Theta(n^{\log_b a} \lg n)$

Case 3: if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, and if

$af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then:

$T(n) = \Theta(f(n))$

regularity condition

Master Method

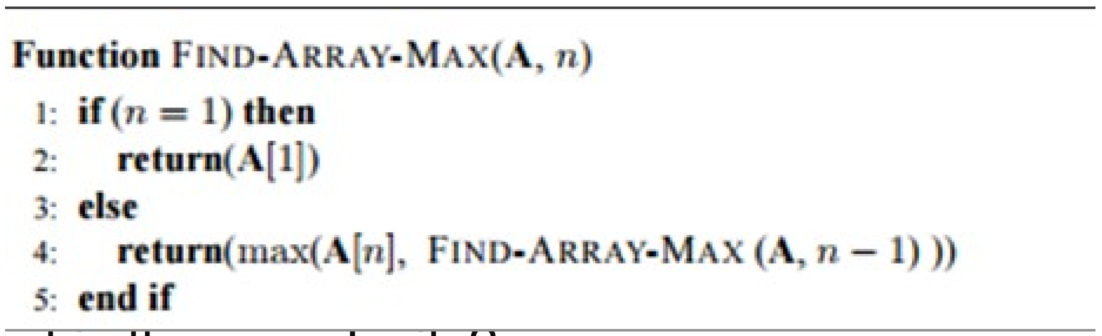
$$T(n) = 9T(n/3) + n$$

$$a = 9, \quad b = 3, \quad f(n) = n$$

$$\left.\begin{aligned} & n^{\log_3 9} = n^2 = \Theta(n^2) \\ & f(n) = n = O(n^{\log_3 9 - \varepsilon}), \ \varepsilon = 1 \end{aligned}\right\} \text{Case 1}$$

$$T(n) = \Theta(n^2)$$

Example: Recursive Find-Array-Max



```
Function FIND-ARRAY-MAX(A, $n$)
if ($n = 1$) then
    return(A[1])
else
    return(max(A[$n$], FIND-ARRAY-MAX (A, $n - 1$) ))
end if
```

What is the complexity?

$T(n) = T(n-1) + c \quad n > 1$

$T(1) = 0$

$f(n) = c = \Theta(1)$ so $d = 0$, $a = 1$ and $b = 1$

So $T(n) = \Theta(n^{0+1}) = \Theta(n)$

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

```
$j \leftarrow 0$
while ($j < n$) do
    $j \leftarrow j+2$
    $z \leftarrow z+1$
```

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 $k \leftarrow 0$ to $n$ do
    for $j \leftarrow 0$ to $k$ do
        $z \leftarrow z+1$
```

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

$$\begin{aligned}&\textstyle\sum_{k=0}^{n}(k+1)\\ &= 1+2+\cdots+n+(n+1)\\ &= \tfrac{(n+1)(n+2)}{2} \in \Theta(n^2).\end{aligned}$$

```
$i \leftarrow n$
while ($i > 1$) do
    $i \leftarrow \lfloor i/2 \rfloor$
    $z \leftarrow z+1$
```

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

```
$j \leftarrow 0$
while ($j < n$) do
    $j \leftarrow j + 2$
    $z \leftarrow z + 1$
```

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 $k \leftarrow 0$ to $n$ do
    for $j \leftarrow 0$ to $k$ do
        $z \leftarrow z + 1$
```

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

$$\begin{aligned} &\textstyle\sum_{k=0}^{n}(k+1) \\ &\quad = 1 + 2 + \cdots + n + (n+1) \\ &\quad = \tfrac{(n+1)(n+2)}{2} \in \Theta(n^2). \end{aligned}$$

```
$i \leftarrow n$
while $(i > 1)$ do
    $i \leftarrow \lfloor i/2 \rfloor$
    $z \leftarrow z + 1$
```

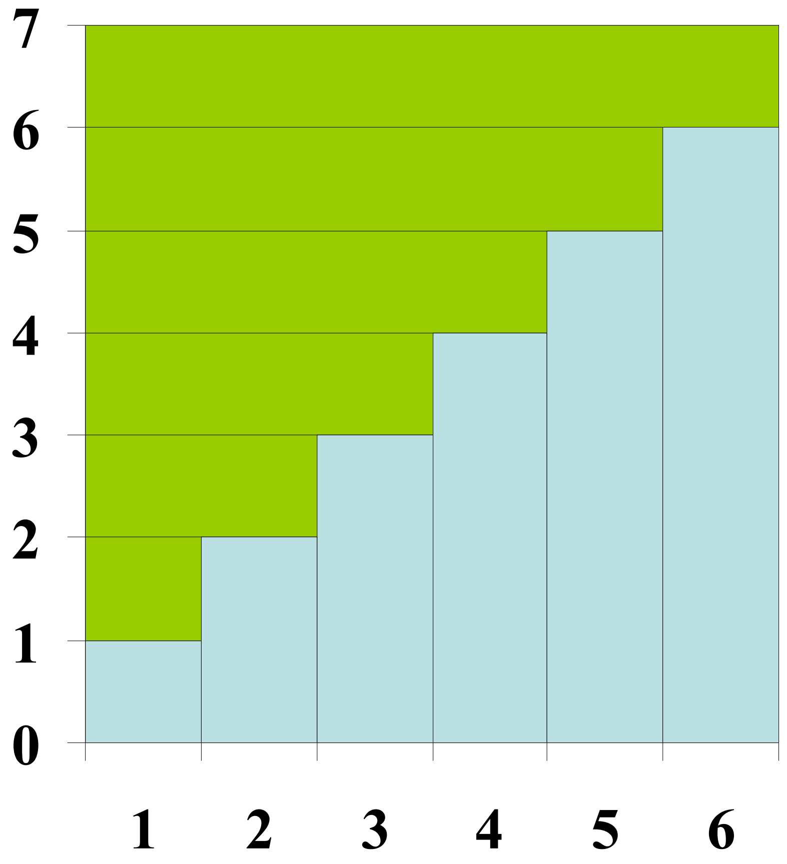
AN ANSWER. Since i takes on the sequence of values $n = n/2^0$, $\lfloor n/2 \rfloor = \lfloor n/2^1 \rfloor$, $\lfloor n/4 \rfloor = \lfloor n/2^2 \rfloor, \ldots, \lfloor n/2^j \rfloor$ until i is ≤ 1. The smallest value of i such that $n/2^i \leq 1$ is $\lceil \log_2 n \rceil$. So there are $(1 +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

Example: Prefix Averages

```
Algorithm prefixAverages1(X, n)
    Input array X of n integers
    Output array A of prefix averages of X      #operations
    A ← new array of n integers                 n
    for i ← 0 to n - 1 do                       n
        s ← X[0]                                n
        for j ← 1 to i do                       1 + 2 + …+ (n - 1)
            s ← s + X[j]
        A[i] ← s / (i + 1)                      n
    return A                                    1
```

Arithmetic Progression

- The running time of *prefixAverages1* is $\Theta(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm *prefixAverages1* runs in $\Theta(n^2)$ time



Prefix Averages 2

The following algorithm computes prefix averages in linear time by keeping a running sum

```
Algorithm prefixAverages2(X, n)
    Input array X of n integers
    Output array A of prefix averages of X          #operations
    A ← new array of n integers                      n
    s ← 0                                            1
    for i ← 0 to n - 1 do                            n
        s ← s + X[i]                                 n
        A[i] ← s / (i + 1)                           n
    return A                                         1
```

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

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 $p_1, p_2, \dots p_n$ each with a duration $d_1, d_2, \dots d_n$ (in days) and you receive the fee $f_1, f_2, \dots, f_n$ (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.

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, .. p_n each with a duration d_1, d_2, .. d_n (in days) and you receive the fee f_1, f_2, .., 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 d = 1 to D {
          If (di > D) {
                  OPT(i, d) = OPT( i-1, d)    // not enough time to complete project i }
          Else {
                  OPT(i, d) = max ( OPT( i-1, d),    // Don't complete project i
                                    OPT( i-1, d-di) + fi  ) //  Complete project I and earn fee fi
                                  )
          }
  }
}
```

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

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

$O(nD)$ or $\Theta(nD)$

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 $8 = 1 + (2 \times 3) + 1$, and given the list 2, 2, 1, 3, 2, 1, 2, 2, 1, 2 the product sum is $19 = (2 \times 2) + 1 + (3 \times 2) + 1 + (2 \times 2) + 1 + 2$.

a) Compute the product-sum of 1, 4, 3, 2, 3, 4, 2.

b) Give the optimization formula OPT[j] for computing the product-sum of the first j elements.

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 $8 = 1 + (2 \times 3) + 1$, and given the list 2, 2, 1, 3, 2, 1, 2, 2, 1, 2 the product sum is $19 = (2 \times 2) + 1 + (3 \times 2) + 1 + (2 \times 2) + 1 + 2$.

a) Compute the product-sum of 1, 4, 3, 2, 3, 4, 2.

b) Give the optimization formula OPT[j] for computing the product-sum of the first j elements.

c) What is the running time of an algorithm that computes the maximum sum.