Lakehead
UNIVERSITY

COMP 4433: Algorithm Design and Analysis

Dr. Y. Gu

Jan. 25, 2023 (Lecture 4)

Divide and Conquer (D&C) (More Examples)

Cases

- Merge Sort
- Quick Sort
- **Maximum Subarray Problem**
- **Strassen's Algorithm**

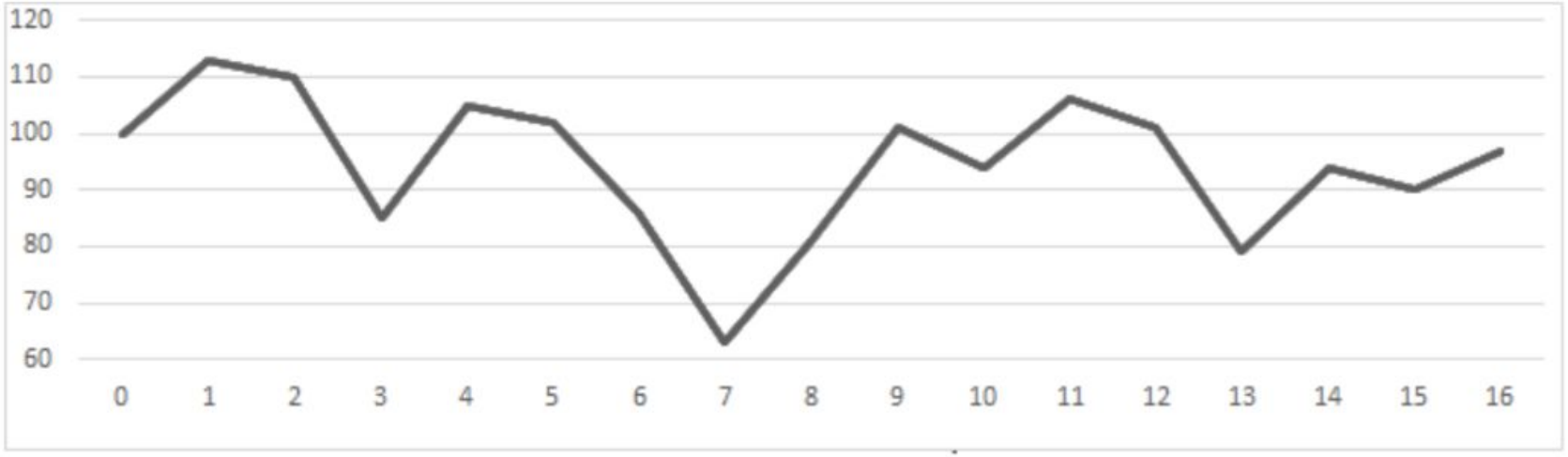
Key Aspects

The divide and conquer paradigm involves three steps at each level of the recursion:

- Divide: break the problem into a number of subproblems that are smaller instances of the same problem.
- Conquer: solve the subproblems recursively or straightforward if the subproblem sizes are small enough.
- Combine: solve the problem by combining the solutions of subproblems.

The Maximum-Subarray Problem

Example: The Figure below shows the price of a stock A over a 17 day period. Our goal is to maximize the profit. Suppose we are only allowed to buy once and sell once during this period, what are the best dates?



Discussion

In general, it is not necessary that buy the stoke at the lowest price or sell the at the highest price will make the best profit. To simplify the discussion, we want to find two days from the stock graph to get highest profit.

A straightforward way is calculating all the possibilities of the buying and selling days. Using this method, we need to compute n*(n-1)/2 pair of values.

Therefore the rough time complexity is $\Omega(n^2)$.

A Better Way

Instead of computing the subtraction of the values of two days, we use the daily changes and compute the maximum sum of a subarray.

Below we record the daily changes of the value of the stoke at the third row. The problem changed to maximum subarray problem.

The maximum subarray problem is **a task to find the series of contiguous elements with the maximum sum in any given array**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

daily changes

Maximum Subarray

Now we consider how to use the divide-and-conquer method to solve the maximum subarray problem. Suppose we have already found the maximum subarray. Then the subarray must be one of the following cases.

1. The subarray sets in the first half (left) of the original array.

2. The subarray sets in the second half (right) of the original array.

3. The subarray sets across the middle of the original array.

Maximum Subarray

- If we can find the maximum subarrays of the above three cases, then we will be easily find the solution.
- The cases 1 and 2 is the problem similar to the original problem, but the data size is half of that of the original problem. This is the main idea about dividing the problem.
- To conquer the problems, we need to solve the case 3.

Case 3 Discussion

Let A[low, . . . , high] be an array and the midpoint of the array is A[mid].

Assume that we have found such a subarray A[i, . . . , mid, mid + 1, . . . , j], which is the maximum subarray of the array A:

1. Then, it must be the case that A[i, . . . , mid] is greater than any subarray with the form A[t, . . . , mid] for any t, where low ≤ t ≤ mid;

How can we do it? Any thought here?

2. and, A[mid + 1, . . . , high] is greater than any subarray with the form A[mid + 1, . . . , t] for any t, where mid + 1 ≤ t ≤ high.

```
procedure Find-Max-Crossing-Subarray(A, low, mid, high)
1:    left-sum = −∞
2:    sum = 0
3:    for i = mid downto low
4:          sum = sum +A[i]
5:          if (sum > left-sum)
6:                left-sum = sum
7:                max-left = i
8:    right-sum = −∞
9:    sum = 0
10:   for j = mid + 1 to high
11:         sum = sum +A[j]
12:         if (sum > right-sum)
13:               right-sum = sum
14:               max-right = j
15:    return (max-left, max-right, left-sum + right-sum)
```

Running Time of Find-Max-Crossing-Subarray

In this procedure, the main running time is used by two for loops, one is the lines 3-7 and another is the lines 10 - 14. Other lines are for initializing variables and take constant time.

In each loop, an iteration also takes a constant time. Therefore we can compute the running time by counting the number of iterations.

In the first loop, it takes $mid - low + 1$ iterations and the second loop takes $high - mid$ iterations.

Suppose the size of the array is n. Then the total number of iterations is $(mid - low + 1) + (high - mid) = high - low - 1 = n$.

It is easy to prove that the running time for this procedure is $\Theta(n)$.

Pseudocode of Find-Maximum-Subarray

```
procedure Find-Maximum-Subarray(A, low, high)
1:    if (high == low)  // A only has one element
2:          return (low, high, A[low])
3:    else
4:          mid = ⌊(low + high)/2⌋
5:          (left-low, left-high, left-sum) = Find-Maximum-Subarray (A, low, mid)
6:          (right-low, right-high, right-sum) =
                        Find-Maximum-Subarray (A, mid + 1, high)
7:          (cross-low, cross-high, cross-sum) =
                        Find-Max-Crossing-Subarray (A, low, mid, high)
8:    if (left-sum ≥right-sum and left-sum ≥ cross-sum)
9:          return (left-low, left-high, left-sum)
10:   else if (right-sum ≥ left-sum and right-sum ≥ cross-sum)
11:         return (right-low, right-high, right-sum)
12:   else
13:         return (cross-low, cross-high, cross-sum)
```

Running Time of Find-Maximum-Subarray

- Lines 1 - 2 treats the base case, in which the array has one element. Since the recursion methods are used, it is important that there is a stop point of each recursion in the procedure. The base cases take constant running time.
- Lines 8-13 returns the maximum subarray among the three subarrays found. It also takes constant running time.
- Lines 5 and 6 use recursive methods. Suppose the running time for Find-Maximum-Subarray is T(n), where n is the size of A. Then line 5 and line 6 needs running time 2T(n/2).
- We already know that line 7 takes $\Theta(n)$ running time.

Therefore, we have …

Running Time of Find-Maximum-Subarray

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n). \end{aligned}$$

In general we have the running time $T(n)$ for Find-Maximum-Subarray

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

What is the computational complexity of this algorithm represented in Θ?

Strassen's Algorithm – Problem

Suppose we have two $n \times n$ matrices $A(a_{i,j})$ and $B(b_{i,j})$. Then the element $c_{i,j}$ of the product $C = A \cdot B$ is defined as:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}.$$

Using a straightforward way to compute the matrices multiplication will need $\Theta(n^3)$ running time.

Straight Algorithm

```
procedure Square-Matrix-Multiply(A, B)
1:    n = A.rows
3:    let C be a new n × n matrix
4:    for i = 1 to n
5:          for j = 1 to n
6:                c(i,j) = 0
7:                for k = 1 to n
8:                      c(i,j) = c(i,j) + a(i,k) · b(k,j)
9:    return C
```

Simple Divide-and- Conquer

Consider using divide-and-conquer.

Suppose two n × n matrices A and B are given and C = A · B.

We partition these matrices into four n/2 × n/2 matrices.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Since $C = A \cdot B$, we have

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

Simple Divide-and- Conquer

$$
\begin{aligned}
C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\
C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\
C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\
C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}
\end{aligned}
$$

Suppose the running time for the multiplying two n × n matrices is T(n). Then the running time for the 8 multiplications of the n/2 × n/2 is 8T(n/2).

We also need to do the matrices additions. Adding two n/2 × n/2 matrices need $n^2/4$ additions. So the total addition time is $\Theta(n^2)$.

We have the recurrence equation of the running time:

$$
T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}
$$

Strassen's Algorithm

Strassen's method improves the above method by reducing the 8 multiplications of sub-matrices to 7.

The Strassen's method has **four** steps:

1. Divide the matrices A, B and C into n/2 × n/2 submatrices as in the previous simple divide-and-conquer method.
2. Create 10 matrices by addition as follows:

$S_1 = B_{12} - B_{22}$ $\quad$ $S_2 = A_{11} + A_{12}$

$S_3 = A_{21} + A_{22}$ $\quad$ $S_4 = B_{21} - B_{11}$

$S_5 = A_{11} + A_{22}$ $\quad$ $S_6 = B_{11} + B_{22}$

$S_7 = A_{12} - A_{22}$ $\quad$ $S_8 = B_{21} + B_{22}$

$S_9 = A11 - A21$ $\quad$ $S_{10} = B_{11} + B_{12}$

Strassen's Algorithm

3. Do 7 multiplications of submatrices:

$P_1 = A_{11} \cdot S_1$ $\quad$ $P_2 = S_2 \cdot B_{22}$

$P_3 = S_3 \cdot B_{11}$ $\quad$ $P_4 = A_{22} \cdot S_4$

$P_5 = S_5 \cdot S_6$ $\quad$ $P_6 = S_7 \cdot S_8$

$P_7 = S_9 \cdot S_{10}$

4. Compute the matrix C as follows:

$C_{11} = P_5 + P_4 - P_2 + P_6$

$C_{12} = P_1 + P_2$

$C_{21} = P_3 + P_4$

$C_{22} = P_5 + P_1 - P_3 - P_7$

Running Time of Strassen's Algorithm

- Step 1 uses constant time, because we can just compute the index.
- Step 2 performs 10 matrices addition each of $n^2/4$ number additions. Therefore step 2 needs 10 $n^2/4$ additions, which is $\Theta(n^2)$.
- Step 3 needs running time 7T(n/2).
- Step 4 calculates 8 submatrices additions, which is also $\Theta(n^2)$.

So the running time for the Strassen's method is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

Comparing Running Time

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

Let a= 8, b = 2, ε=0.5, then, $f(n)=n^2= O(n^{2.5})$, according to Case 1, $T(n)= \Theta(n^3)$.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

We claim that $T(n)= O(n^{\lg 7})$.

After class exercise: proof the claim.

Asymptotic Notation Relationships (More Examples)

Examples

- 7n-2

  7n-2 is O(n)

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$

- 3n3 + 20n2 + 5

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

- 3 log n + log log n

  3 log n + log log n is O(log n)

  need $c > 0$ and $n_0 \geq 1$ s.t. $3 \log n + \log \log n \leq c \bullet \log n$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 2$

Things you need to know

Things you should know. There is no need to memorize the bounds on n; you can always work them if you have to.

- $1 \leq \log(n)$ if $n \geq 10$.
- $\log n \leq n^{\alpha}$ if $\alpha > 0$ (the bound on n depends on α).
- $n^{\alpha} \leq t^{n}$ for $\alpha > 0$ and $t > 1$ (the bound on n depends on α and t).
- $s^{n} \leq t^{n}$ if $1 \leq s \leq t$ and $n \geq 1$.
- $t^{n} \leq n!$ (the bound on n depends on t).
- $n! \leq n^{n}$ if $n \geq 1$.

Lakehead UNIVERSITY

Dynamic Programming

Introduction

- Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems.
- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems.
- “Programming” here means “planning”.
- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances **once**
  - record solutions in a table
  - extract solution to the initial instance from that table

Introduction

- Dynamic programming is a way of improving on inefficient divide-and-conquer algorithms.
- By “inefficient”, we mean that the same recursive call is made over and over.
- If same subproblems is solved several times, we can use **table** to store result of a subproblem the first time it is computed and thus never have to recompute it again.
- Dynamic programming is applicable when the subproblems are dependent, that is, when subproblems share sub-sub-problems.
- “Programming” refers to a tabular method.

DP v.s. D&C

Divide & Conquer	Dynamic Programming
1. Partitions a problem into independent smaller sub-problems	1. Partitions a problem into overlapping sub-problems
2. Doesn't store solutions of sub-problems. (Identical sub-problems may arise - results in the same computations are performed repeatedly.)	2. Stores solutions of sub-problems: thus avoids calculations of same quantity twice
3. Top down algorithms: which logically progresses from the initial instance down to the smallest sub-instances via intermediate sub-instances.	3. Bottom up algorithms: in which the smallest sub-problems are explicitly solved first and the results of these used to construct solutions to progressively larger sub-instances

An Example

- **Rod cutting problem**

The rod cutting problem is the following. Given a rod of length n inches and a table of price p_i for i = 1, . . . , n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. The following is an example of price table.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	19	17	17	20	24	30

An Example

- **Rod cutting problem**

For n = 4, we may cut as: (1, 1, 1, 1),(1, 1, 2),(2, 2),(1, 3),(4),
the correspondent prices are: 4, 7, 10, 9, 9, respectively.

• So the optimal revenue is cutting the 4-inch rod into two 2-inch pieces.

An Example

- **Rod cutting problem**

For n = 4, we may cut as: (1, 1, 1, 1),(1, 1, 2),(2, 2),(1, 3),(4),
the correspondent prices are: 4, 7, 10, 9, 9, respectively.

• So the optimal revenue is cutting the 4-inch rod into two 2-inch pieces.

An Example

By inspection, we can obtain the optimal decomposition as follows.

r_1 = 1 from solution 1 = 1 (no cuts)
r_2 = 5 from solution 2 = 2 (no cuts)
r_3 = 8 from solution 3 = 3 (no cuts)
r_4 = 10 from solution 4 = 2 + 2
r_5 = 13 from solution 5 = 2 + 3
r_6 = 17 from solution 6 = 6 (no cut)
r_7 = 18 from solution 7 = 1 + 6 or 7 = 2 + 2 + 3
r_8 = 22 from solution 8 = 2 + 6
r_9 = 25 from solution 9 = 3 + 6
r_{10} = 30 from solution 10 = 10 (no cuts)

An Example

In general, for a rod of length n, we can consider 2^{n-1} different cutting ways, since we have an independent option of cutting or not cutting at distance i inches from one end.

Suppose an optimal solution cuts the rod into k pieces with lengths $i_1, i_2, \ldots, i_k$ that

$$n = i_1 + i_2 + \cdots + i_k$$

and the corresponding optimal revenue is

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}.$$

Our purpose is to compute r_n for given n and p_i, i = 1, . . . , n. When we consider dividing the problem, we can use the following method:

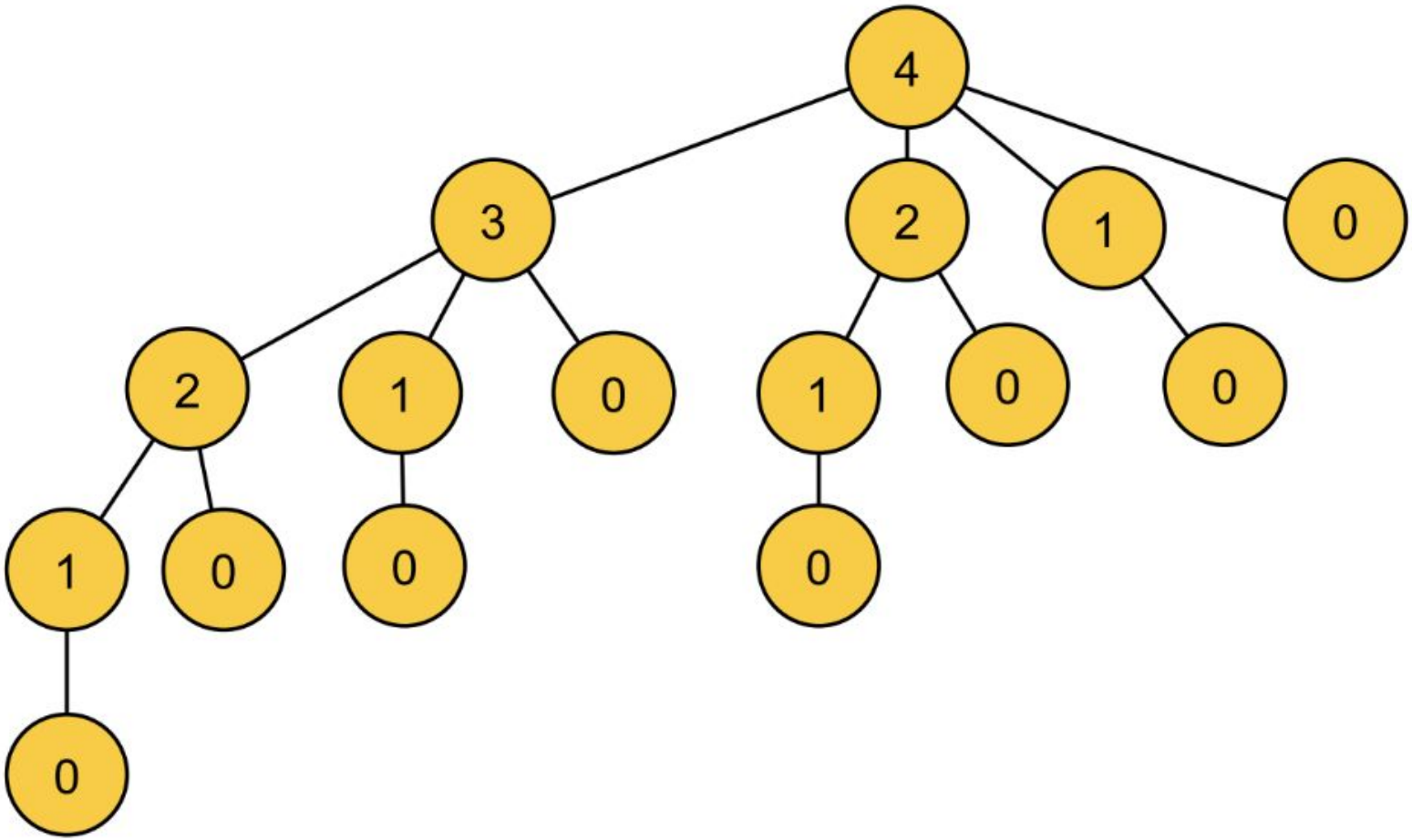
$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1)$$

The first case is no cutting. The other cases consider optimal substructure: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

One approach:

```
procedure Cut-Rod(p, n)
1:   if n == 0 then
2:         return 0
3:   q = −∞
4:   for i = 1 to n do
5:         q = max(q, p[i]+Cut-Rod(p, n − i))
6:   return q
```

It is inefficient and why?


4
3
2
1
0
2
1
0
1
0
0
1
0
0
0
0