Lecture 3: The Maximum Subarray Problem

Version of Feb 2, 2019

Divide-and-Conquer

Divide-and-conquer.

- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

Most common pattern.

- Break up problem of size n into two equal parts of size $\frac{1}{2}n$.
- Solve two parts recursively.
- Combine two solutions into overall solution.

Techniques needed.

- Algorithm uses recursion.
- Analysis uses recurrences.

Previous Example Seen

- Merge Sort

The Maximum Subarray Problem

Input: Profit history of a company. Money earned/lost each year.

Year	1	2	3	4	5	6	7	8	9
Profit (M$)	3	2	1	-7	5	2	-1	3	-1

Problem: Find the span of years in which the company earned the most

Answer: Year 5-8 , 9 M$

Formal definition:

Input: An array of numbers $A[1 \dots n]$, both positive and negative

Output: Find the maximum $V(i,j)$, where $V(i,j) = \sum_{k=i}^{j} A[k]$

A brute-force algorithm

Idea: Calculate the value of $V(i,j)$ for each pair $i \le j$ and return the maximum value.

```
V_max ← A[1]
for i ← 1 to n do
    for j ← i to n do
        // calculate V(i,j)
        V ← 0
        for k ← i to j do
            V ← V + A[k]
        if V > V_max then V_max ← V
return V_max
```

Running time: $\Theta(n^3)$

Intuition: Calculating value of $\Theta(n^2)$ arrays, each one, on average, $\Theta(n/2)$ long.

A data-reuse algorithm

Idea:

- Don't need to calculate each $V(i,j)$ from scratch.
- Exploit the fact: $V(i,j) = V(i,j-1) + A[j]$

```
V_max ← A[1]
for i ← 1 to n do
    V ← 0
    for j ← i to n do
        // calculate V(i,j)
        V ← V + A[j]
        if V > V_max then V_max ← V;
return V_max
```

Running time: $\Theta(n^2)$

Intuition: *Fix starting point i.*

Calculating $V(i,j)$ *from* $V(i,j-1)$ *requires only* $\Theta(1)$ *time.*

=> $\Theta(n^2)$ in total.

A divide-and-conquer algorithm

Year	1	2	3	4	5	6	7	8	9
Profit (M$)	3	2	1	-7	5	2	-1	3	-1

Idea:

- Cut the array into two halves
- All subarrays can be classified into three cases:
  - Case 1: entirely in the first half
  - Case 2: entirely in the second half
  - Case 3: crosses the cut
- Largest of three cases is final solution
- The optimal solutions for case 1 and 2 can be found recursively.
- Only need to consider case 3.
- Compare with merge sort:
  If we can solve case 3 in linear ($O(n)$) time,
  => whole algorithm will run in $\Theta(n \log n)$ time.

$$T(n) = 2T(n/2) + n \quad \Rightarrow \quad T(n) = \Theta(n \log n)$$

Solving case 3

Year	1	2	3	4	5	6	7	8	9
Profit (M$)	3	2	1	-7	5	2	-1	3	-1

Idea:

- To solve problem in subarray $A[p..r]$; Let $q = \lfloor (p+r)/2 \rfloor$
- Any case 3 subarray must have $p \le q < r$
- Such a subarray can be divided into two parts

  $A[i..q]$ and $A[q+1..j]$, for some i and j
- Just need to maximize each of them separately

To maximize $A[i..q]$ and $A[q+1, j]$:

- Let $i = i'$, $j = j'$ be the indices that

  maximize the values $A[i..q]$ and $A[q+1, j]$:
- i', j' can be found by using separate linear scans to left and right of q

=> $A[i'..j']$ has largest value of all subarrays that cross q

The complete divide-and-conquer algorithm

```
MaxSubarray(A,p,r):
if p = r then return A[p]
q ← ⌊(p + r)/2⌋
M_1 ← MaxSubarray(A,p,q)              % MAX Left Half
M_2 ← MaxSubarray(A,q+1,r)            % MAX Right Half
L_m ← −∞, R_m ← −∞
V ← 0
for i ← q downto p                    % MAX Left
      V ← V + A[i]                    % starting at q
      if V > L_m then L_m ← V
V ← 0
for i ← q+1 to r                      % MAX Right
      V ← V + A[i]                    % starting at q
      if V > R_m then R_m ← V
return max{M_1, M_2, L_m + R_m}

First call: MaxSubarray(A,1,n)
```

Analysis:

- Recurrence:
  $T(n) = 2T(n/2) + n$
- => $T(n) = \Theta(n \log n)$

A linear-time algorithm?

Define: $X[i] = A[1] + \cdots + A[i]$

Set: $X[0] = 0$

Year	1	2	3	4	5	6	7	8	9
Profit (M$)	-3	2	1	-4	5	2	-1	3	-1
$X[i]$	-3	-1	0	-4	1	3	2	5	4

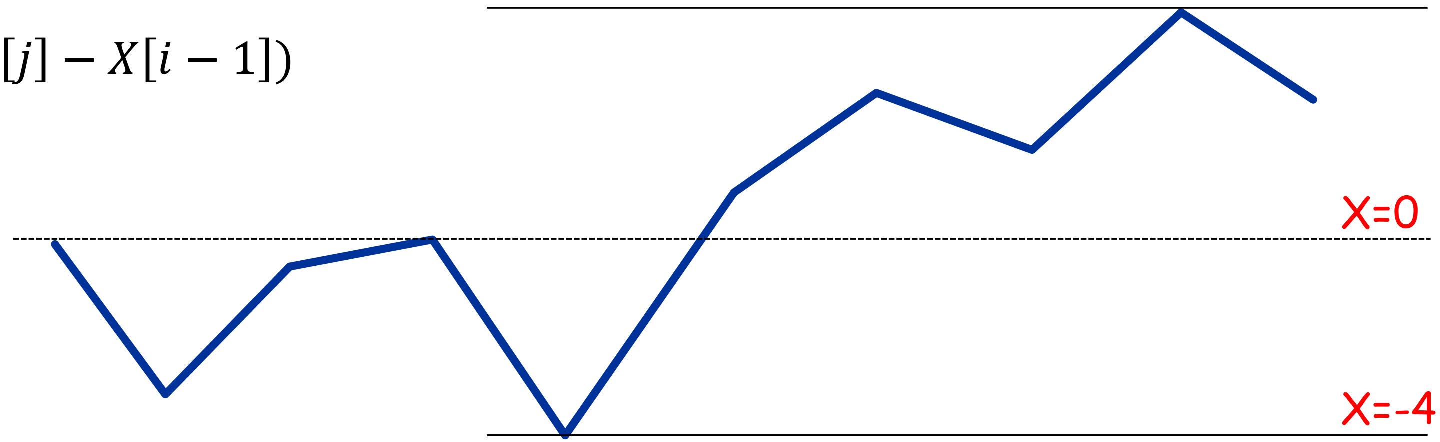
Observations:

- $V(i,j) = \sum_{k=i}^{j} A[k] = X[j] - X[i-1]$
- For fixed j, finding largest $V(i,j)$ is same as
  knowing the index i, $i \leq j$ for which $X[i-1]$ is smallest
- Finding this for each j, lets us find overall largest $V(i,j)$

A linear-time ($\Theta(n)$) algorithm?

Define: $X[i] = A[1] + \cdots + A[i]$

Goal: Find $\max_{i \le j}(X[j] - X[i-1])$



Year	1	2	3	4	5	6	7	8	9
Profit (M$)	-3	2	1	-4	5	2	-1	3	-1
$X[i]$	-3	-1	0	-4	1	3	2	5	4

Algorithm:

- For each j, needs to know $i \le j$ that minimizes $X[i-1]$
  (i.e., maximizes $X[j] - X[i-1]$)
  - (Then maximize over all j)
- Algorithm increases j by +1 each step
- Keeps track of smallest $X[i]$ so far
  - Could be old smallest one, or it could be current $X[j]$

The linear-time algorithm

Clever Algorithm

$V_{max} \leftarrow -\infty, X_{min} = 0$
$X \leftarrow 0, V \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    $V \leftarrow V + A[i]$
    if $V > V_{max}$ **then** $V_{max} \leftarrow V$
    $X \leftarrow X + A[i]$
    if $X < X_{min}$ **then**
        $X_{min} \leftarrow X$
        $V \leftarrow 0$
return V_{max}

Just Showing Off

Even "simpler":

$V_{max} \leftarrow -\infty, V \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    $V \leftarrow V + A[i]$
    if $V > V_{max}$ **then** $V_{max} \leftarrow V$
    if $V < 0$ **then** $V \leftarrow 0$
return V_{max}

Observation:

- $X < X_{min}$ iff $V < 0$
  - Because $V = X - X_{min}$
- No need to actually store X!

At any time i

- X_{min} keeps track of smallest $X[i]$ seen so far.
- $V = X[i] - X_{min}$
- X stores X[i]

Maximum Sub-Array Algorithm Design

- $\Theta(n^3)$ Algorithm
  - Directly from problem definition
- $\Theta(n^2)$ Algorithm
  - Simple reuse of information
  - Trivial observation
- $\Theta(n \log n)$ Algorithm
  - Application of algorithm design principles
  - Divide-and-conquer
  - Expected of you after taking this class
- $\Theta(n)$ Algorithm
  - In beginning, people thought that $\Theta(n \log n)$ was best possible
  - Required ah-ha moment through re-visualization of problem
  - More art than science (although knowing the science led to the art)