Lecture 3: The Maximum Subarray Problem

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.

A simple divide-and-conquer algorithm: Binary search

Input: An array A of elements in sorted order, and an element x.

Output: Return the position of x if it exists; otherwise output nil.

4	7	10	15	19	20	42	54	87	90

```
BinarySearch(A,p,r,x):
if p > r then return nil
q ← ⌊(p + r)/2⌋
if A[q] = x return q
if x < A[q] then BinarySearch(A,p,q − 1,x)
else BinarySearch(A,q + 1,r,x)

First call: BinarySearch(A,1,n,x)
```

Recurrence: $T(n) = T(n/2) + 1$, which solves to $T(n) = \Theta(\log n)$.

Note: Unlike merge sort, this algorithm may terminate faster than $\Theta(\log n)$, but the worst-case running time is still $\Theta(\log n)$

The Maximum Subarray Problem

Input: Profit history of a company of the years.

Year	1	2	3	4	5	6	7	8	9
Profit (M$)	-3	2	1	-4	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 $i \leq j$ 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: $\Theta(n^3)$

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: $\Theta(n^2)$

A divide-and-conquer algorithm

Year	1	2	3	4	5	6	7	8	9
Profit (M$)	-3	2	1	-4	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: cross the cut
- The optimal solution 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 time, the whole algorithm will run in $\Theta(n \log n)$ time.

Solving case 3

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

Idea:

- Let $q = \lfloor (p+r)/2 \rfloor$
- Any case 3 subarray must have starting position $\leq q$, and ending position $\geq q+1$
- 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

Maximize $A[i..q]$ and $A[q+1, j]$: The data-reuse idea again!

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)
M_2 ← MaxSubarray(A,q + 1,r)
L_m ← −∞, R_m ← −∞
V ← 0
for i ← q downto p
      V ← V + A[i]
      if V > L_m then L_m ← V
V ← 0
for i ← q + 1 to r
      V ← V + A[i]
      if V > R_m then R_m ← V
return max{M_1, M_2, L_m + R_m}

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

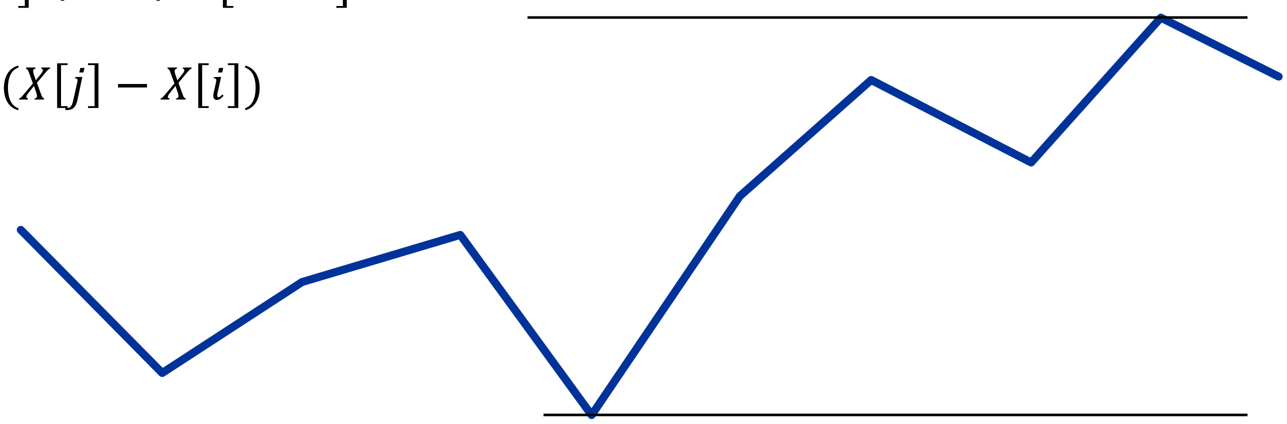
Analysis:

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

A linear-time algorithm?

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

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



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

Observations:

- For some i, suppose $X[i]$ is the lowest point before i (including).
- The optimal interval cannot cross i
  - Otherwise, moving the starting point to i would make it better
- The optimal solution must start from such an i
  - Otherwise, could move the starting point to a lower point

The linear-time algorithm

```
V_max ← −∞, X_min = 0
X ← 0, V ← 0
for i ← 1 to n do
    V ← V + A[i]
    if V > V_max then V_max ← V
    X ← X + A[i]
    if X < X_min then
        X_min ← X
        V ← 0
return V_m
```

Even simpler:

```
V_max ← −∞, V ← 0
for i ← 1 to n do
    V ← V + A[i]
    if V > V_max then V_max ← V
    if V < 0 then V ← 0
return V_max
```

Observation:

- $X < X_{min}$ iff $V < 0$
- No need for X!