CS 344: Design & Analysis of Algorithms

Lecture 3

Sep 10, 2019

Algorithm design

We can design algorithms a few ways:

- incremental – sort up to $j - 1$, then sort up to j
- divide and conquer

Divide and conquer

- divide: split problem into subproblems of the same type
- conquer: solve the subproblems recursively
- combine: combine the results into a solution for the original problem

Merge sort

Given an unsorted array, if we could somehow separate it into two sorted arrays, we could then merge these two to get a final sorted array.

- Break it into two arrays
- An array of size 1 is already sorted

Merge sort

Suppose we have the following unsorted array:

45	-6	73	28	5	-1	23	42

Merge sort

Repeatedly divide

45 -6 73 28 5 -1 23 42

45 -6 73 28   5 -1 23 42

Merge sort

Repeatedly divide

45 -6 73 28 5 -1 23 42

45 -6 73 28 | 5 -1 23 42

45 -6 | 73 28 | 5 -1 | 23 42

Merge sort

Repeatedly divide

45 -6 73 28 5 -1 23 42

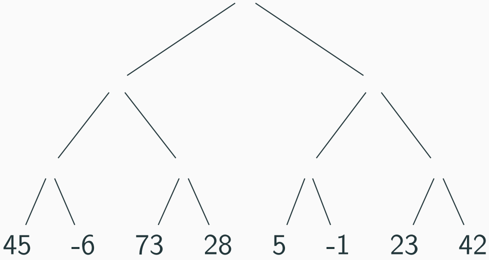
45 -6 73 28 | 5 -1 23 42

45 -6 | 73 28 | 5 -1 | 23 42

45 | -6 | 73 | 28 | 5 | -1 | 23 | 42

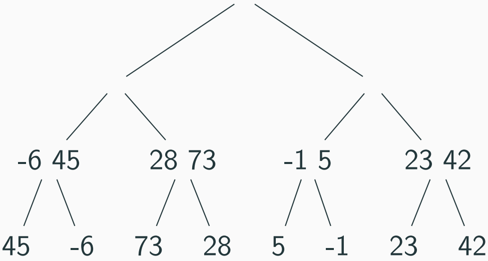
Merge sort

Now merge:



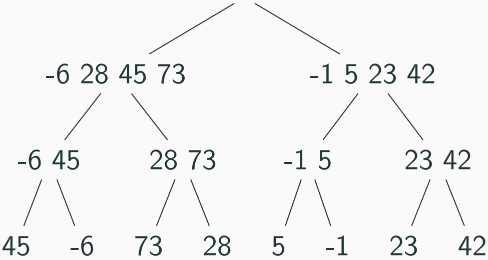
Merge sort

Now merge:



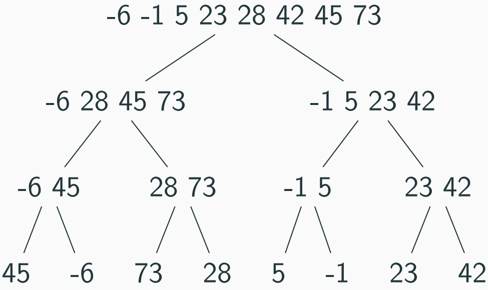
Merge sort

Now merge:



Merge sort

Now merge:



Merge sort

What is the running time of merge sort?

- Divide n elements into two sets of $n/2$ elements and solve
- Merge the results

Merge sort

How long does it take to merge two sorted $n/2$ lists?

1	5	10	20

3	4	7	42

1	3	4	5	7	10	20	42

Merge sort

What is the running time of merge sort?

$$
\begin{aligned}
T(n) &= 2T(n/2) + n = 2(2T(n/4) + n/2) + n \\
&= 4T(n/4) + 2n = 4(2T(n/8) + n/4) + 2n \\
&= 8T(n/8) + 3n \\
&= ... \\
&= 2^k T(n/2^k) + kn
\end{aligned}
$$

Merge sort

What is the running time of merge sort?

$$T(n) = 2^k T(n/2^k) + kn$$

Let $n = 2^k$:

$$\begin{aligned} T(n) &= nT(n/n) + kn \\ &= nT(1) + kn \\ &= n \cdot 1 + kn \\ &= n + kn \end{aligned}$$

Merge sort

What is the running time of merge sort?

$$T(n) = n + kn$$

To get rid of k, observe that $n = 2^k$ implies $k = \log n$:

$$\begin{aligned} T(n) &= n + (\log n)n \\ &= O(n) + O(n \log n) \\ &= O(n \log n) \end{aligned}$$

Buying stocks

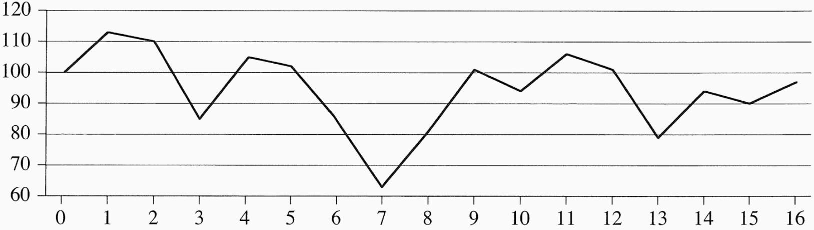
Suppose you invent an AI that correctly predicts the stock market for the next two weeks.

And suppose you want to buy a stock on one day and sell another day.

What are the best days to buy and sell to maximize your profit?

Buying stocks

Stock prices:



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

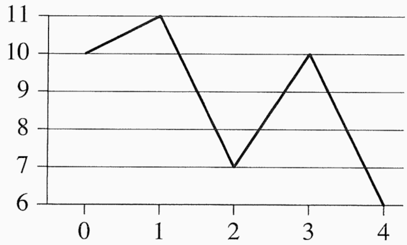
Buying stocks

First attempt:

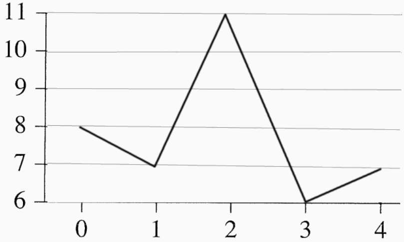
- Find the date of the stock's lowest price, buy then
- Find the highest later price, sell then

Buying stocks

But this may not give an optimal result:



Another case where we don't get the optimal result:



Buying stocks

We could solve this via brute force.

- for each day we could buy:
  - for each (later) day we could sell:
    - calculate the profit
    - keep track of the max

Buying stocks

We could solve this via brute force.

```
max = 0
best_i = 0
best_j = 0
for i from 0 to n:
    for j from i+1 to n:
        if a[j] - a[i] > max:
            max = a[j] - a[i]
            best_i = i
            best_j = j
```

Buying stocks

How long would the brute force method take?

- $n-1$ sell dates if we buy on day 1
- $n-2$ sell dates if we buy on day 2
- ...
- 2 sell dates if we buy on day $n-2$
- 1 sell date if we buy on day $n-1$

For each pair (buyDay, sellDay), we do a constant amount of work ($\Theta(1)$).

Buying stocks

How long would the brute force method take?

- There are $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ pairs.
- For each pair (buyDay, sellDay), we do a constant amount of work ($\Theta(1)$).
- $O(n^2)$ total work.

Maximum subarray problem

First let's consider just the price changes each day:

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

Maximum subarray problem

Then we're looking for a subarray with the largest sum:

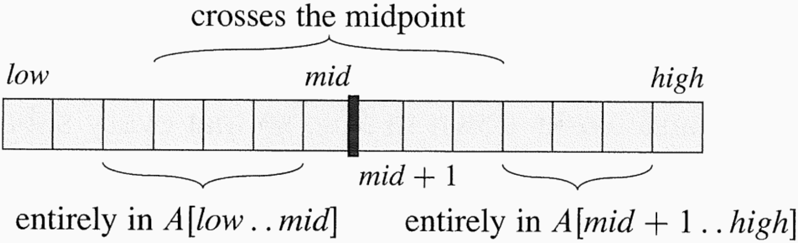
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

Maximum subarray problem

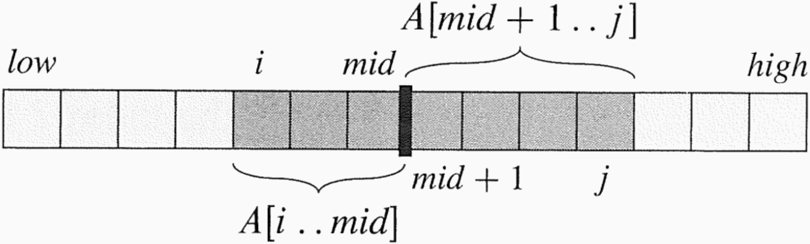
If we split the array in two equal parts, where could the maximum subarray lie?

- Entirely in the first half
- Entirely in the second half
- Partly in each, crossing the midpoint



Maximum subarray problem

If it crosses the midpoint, there must be a part in the left and a part in the right.



Maximum subarray problem

$left\text{-}sum = -\infty$
$sum = 0$
for $i = mid$ **downto** low
    $sum = sum + A[i]$
    if $sum > left\text{-}sum$
        $left\text{-}sum = sum$
        $max\text{-}left = i$
$right\text{-}sum = -\infty$
$sum = 0$
for $j = mid + 1$ **to** $high$
    $sum = sum + A[j]$
    if $sum > right\text{-}sum$
        $right\text{-}sum = sum$
        $max\text{-}right = j$
return $(max\text{-}left, max\text{-}right, left\text{-}sum + right\text{-}sum)$

Maximum subarray problem

if *high* == *low*
    return $(low, high, A[low])$
else $mid = \lfloor (low + high)/2 \rfloor$
    $(\textit{left-low}, \textit{left-high}, \textit{left-sum}) =$
        $\textsc{Find-Maximum-Subarray}(A, low, mid)$
    $(\textit{right-low}, \textit{right-high}, \textit{right-sum}) =$
        $\textsc{Find-Maximum-Subarray}(A, mid + 1, high)$
    $(\textit{cross-low}, \textit{cross-high}, \textit{cross-sum}) =$
        $\textsc{Find-Max-Crossing-Subarray}(A, low, mid, high)$
    if $\textit{left-sum} \geq \textit{right-sum}$ and $\textit{left-sum} \geq \textit{cross-sum}$
        return $(\textit{left-low}, \textit{left-high}, \textit{left-sum})$
    elseif $\textit{right-sum} \geq \textit{left-sum}$ and $\textit{right-sum} \geq \textit{cross-sum}$
        return $(\textit{right-low}, \textit{right-high}, \textit{right-sum})$
    else return $(\textit{cross-low}, \textit{cross-high}, \textit{cross-sum})$

Maximum subarray problem

How long does this take?

- base case: $T(1) = O(1)$
- recursion: 2 subproblems of size $n/2$
- finding the max crossing subarray: $O(n)$

Maximum subarray problem

How long does this take?

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

Same as merge sort! $O(n \log n)$

Solving recurrences

- Substitution method
- Recursion tree method
- Master method

Master theorem

Divide and conquer solves a problem of size n by:

- splitting it into a subproblems of size n/b
- combining the answers in $O(n^d)$ time

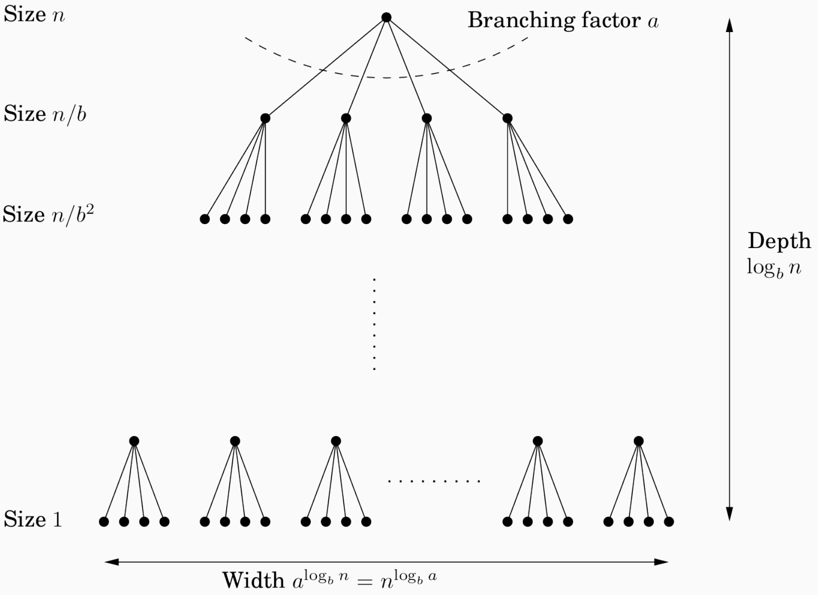
where $a, b, d > 0$

Master theorem

If $T(n) = aT(n/b) + O(n^d)$ and $a > 0, b > 1, d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Master theorem



Master theorem

The work done at each level of the tree is

$$a^k \cdot O\left(\frac{n}{b^k}\right)^d = O(n^d) \cdot \left(\frac{a}{b^d}\right)^k$$

Master theorem

$$O(n^d) \cdot \left(\frac{a}{b^d}\right)^k$$

Taking k from 0 to $\log n$ yields a geometric series with ratio a/b^d:

$$O(n^d) + O(n^d) \cdot \left(\frac{a}{b^d}\right) + O(n^d) \cdot \left(\frac{a}{b^d}\right)^2 + \cdots + O(n^d) \left(\frac{a}{b^d}\right)^{\log_b n}$$

Asymptotics of a geometric series

Suppose $c > 0$ and $g(n) = 1 + c + c^2 + \cdots + c^n$.

Then:

- if $c < 1$, $g(n) = \Theta(?)$
- if $c = 1$, $g(n) = \Theta(?)$
- if $c > 1$, $g(n) = \Theta(?)$

Asymptotics of a geometric series

Suppose $c > 0$ and $g(n) = 1 + c + c^2 + \cdots + c^n$.

Then:

- if $c < 1$, $g(n) = \Theta(1)$
- if $c = 1$, $g(n) = \Theta(n)$
- if $c > 1$, $g(n) = \Theta(c^n)$

Master theorem

$$O(n^d) \cdot \left(\frac{a}{b^d}\right)^k$$

Taking k from 0 to $\log n$ yields a geometric series with ratio a/b^d:

- ratio < 1: use first term, $O(n^d)$
- ratio $= 1$: all $O(\log n)$ terms are $O(n^d)$, so we have $O(n^d \log n)$
- ratio > 1: use last term, $O(n^{\log_b a})$

Changing base of logarithms

We can convert logarithm bases by multiplying by a constant factor:

$$\log_a n = \frac{\log_b n}{\log_b a}$$

so

$$\log_b n = (\log_a n)(\log_b a)$$

Master theorem

$$
\begin{aligned}
n^d \left(\frac{a}{b^d}\right)^{log_b n} &= n^d \left(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\right) \\
&= a^{\log_b n} \\
&= a^{(\log_a n)(\log_b a)} \\
&= n^{\log_b a}
\end{aligned}
$$

Master theorem

And these three cases correspond to the three cases of the master theorem.

If $T(n) = aT(n/b) + O(n^d)$ and $a > 0, b > 1, d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Binary search

Given a sorted array, use binary search to find an element k:

- Divide: search for k in either the left or right half
- Combine: return the result

Binary search

$$T(n) = T(n/2) + O(1)$$

Binary search

$$T(n) = T(n/2) + O(1)$$

Master theorem: $T(n) = aT(n/b) + O(n^d)$

- $a = 1$
- $b = 2$
- $d = 0$

Binary search

If $T(n) = aT(n/b) + O(n^d)$ and $a > 0, b > 1, d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

- $a = 1$
- $b = 2$
- $d = 0$

$$\log_b a = \log_2 1 = 0 = d$$

So we use the second case.

Binary search

If $T(n) = aT(n/b) + O(n^d)$ and $a > 0, b > 1, d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

- $a = 1$
- $b = 2$
- $d = 0$

$$O(n^d \log n) = O(n^0 \log n) = O(\log n)$$

Merge sort

Recall the recurrence for merge sort:

$$T(n) = 2T(n/2) + O(n)$$

To match $T(n) = aT(n/b) + O(n^d)$:

- $a = 2$
- $b = 2$
- $d = 1$

Merge sort

If $T(n) = aT(n/b) + O(n^d)$ and $a > 0, b > 1, d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

- $a = 2$
- $b = 2$
- $d = 1$

$$\log_b a = \log_2 2 = 1 = d$$

Second case again!

Merge sort

If $T(n) = aT(n/b) + O(n^d)$ and $a > 0, b > 1, d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

- $a = 2$
- $b = 2$
- $d = 1$

$$O(n^d \log n) = O(n^1 \log n) = O(n \log n)$$

Matrix multiplication

Recall:

$$A \cdot B = C$$

then c_{ij} is the dot product of row i of A and column j of B.

Matrix multiplication

- How long does the naive multiplication method take?
- Can we view this as a divide and conquer algorithm?

Matrix multiplication

Suppose we decompose the matrices into blocks:

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

Matrix multiplication

Then we can define multiplication as such:

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

So we have these subproblems to compute:

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

Matrix multiplication

$n = A.rows$
let C be a new $n \times n$ matrix
if $n == 1$
$\quad c_{11} = a_{11} \cdot b_{11}$
else partition A, B, and C as in equations (4.9)
$\quad C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11})$
$\quad\quad + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21})$
$\quad C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12})$
$\quad\quad + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22})$
$\quad C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11})$
$\quad\quad + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21})$
$\quad C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12})$
$\quad\quad + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22})$
return C

Matrix multiplication

- Divide into 8 blocks of size $n/2 \times n/2$
- Recursively multiply
- Add (some of) the resulting matrices

Matrix multiplication

Then the time required has the form:

$$T(n) = 8T(n/2) + O(n^2)$$

Matrix multiplication

$$T(n) = 8T(n/2) + O(n^2)$$

Master theorem: $T(n) = aT(n/b) + O(n^d)$

- $a = 8$
- $b = 2$
- $d = 2$

Matrix multiplication

If $T(n) = aT(n/b) + O(n^d)$ and $a > 0, b > 1, d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

- $a = 8$
- $b = 2$
- $d = 2$

$$\log_b a = log_2 8 = 3 > 2 = d$$

Use case 3.

Matrix multiplication

If $T(n) = aT(n/b) + O(n^d)$ and $a > 0, b > 1, d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

- $a = 8$
- $b = 2$
- $d = 2$

$$O(n^{\log_b a}) = O(n^3)$$