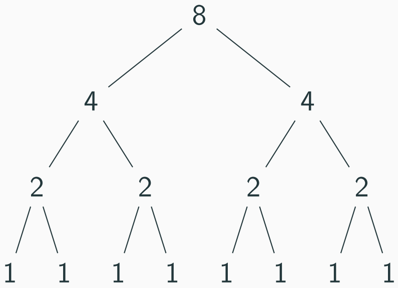
CS 344: Design & Analysis of Algorithms

Lecture 5

Sep 17, 2019

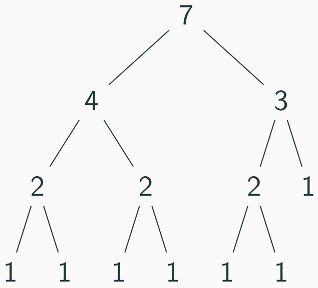
Merge sort



$n = 2^k$, which gave us the recurrence

$$\begin{aligned} T(n) &= 2^k T(n/2^k) + kn \\ &= O(n \lg n) \end{aligned}$$

Merge sort



$n = 2^k + 2c$

Merge sort

Then on the last full level, there are $2^k - c$ leaf nodes and c nodes with size 2.

$$T(n) = (2^k - c) \cdot T(1) + c \cdot T(2) + kn$$

Merge sort

$$
\begin{aligned}
T(n) &= (2^k - c) \cdot T(1) + c \cdot T(2) + kn \\
&= (2^k - c) \cdot 1 + c \cdot 4 + kn \\
&= 2^k + 3c + kn
\end{aligned}
$$

Merge sort

Since $n = 2^k + c$, we have $k = \lg(n - c)$

$$\begin{aligned} T(n) = 2^k + 3c + kn &= 2^{\lg(n-c)} + 3c + \lg(n - c) \cdot n \\ &= n + 2c + n \lg(n - c) \\ &= O(n \lg n) \end{aligned}$$

Maximum subarray problem

	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

Note that any subarray must end at some index i.

Let's consider the problem inductively:

- Suppose we know the maximum subarray ending at index i
- Can we figure out the maximum subarray ending at $i + 1$?

Maximum subarray problem

This is known as Kadane's algorithm.

```python
def maxSubarray(numbers):
    bestSum = 0
    currSum = 0
    for x in numbers:
        currSum = max(0, currSum + x)
        bestSum = max(bestSum, currSum)
    return bestSum
```

Maximum subarray problem

Consider the array 4 -6 7 2 5 -1 3 2:

i	A[i]	current sum	best sum
		0	0
0	4	max(0, 4) = 4	4
1	-6	max(0, -2) = 0	4
2	7	max(0, 7) = 7	7
3	2	max(0, 9) = 9	9
4	5	max(0, 14) = 14	14
5	-1	max(0, 13) = 13	14
6	3	max(0, 16) = 16	16
7	2	max(0, 18) = 18	18

Maximum subarray problem

Kadane's algorithm:

```
def maxSubarray(numbers):
    bestSum = 0
    currSum = 0
    for x in numbers:
        currSum = max(0, currSum + x)
        bestSum = max(bestSum, currSum)
    return bestSum
```

What is the running time?

Substitution method

Another method to solve recurrences is the "substitution method":

- Guess the correct form
- Prove inductively

Substitution method

Given the following recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\lfloor n/2 \rfloor) + n & \text{otherwise} \end{cases}$$

Since it's similar to $2T(n/2) + n$, we could guess that it's also $O(n \lg n)$.

Substitution method

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\lfloor n/2 \rfloor) + n & \text{otherwise} \end{cases}$$

To prove it's $O(n \lg n)$, we must show $T(n) \leq cn \lg n$ for some $c > 0$, for sufficiently large n.

Substitution method

Inductive step: assume this is true for all $m < n$

Then it's true for $m = \lfloor n/2 \rfloor$,

so we have $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$

$$
\begin{aligned}
T(n) &= 2T(\lfloor n/2 \rfloor) + n \\
&\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\
&\leq cn \lg(n/2) + n \\
&= cn \lg n - cn \lg 2 + n \\
&= cn \lg n - cn + n \\
&= cn \lg n - (c-1)n \\
&\leq cn \lg n
\end{aligned}
$$

Substitution method

Base case: we're given $T(1) = 1$, and must show

$$T(1) \leq cn \lg n$$

But this equals $c1 \lg 1 = 0$...

Substitution method

Fortunately, we only need this to hold for $n > n_0$

By the recurrence, $T(2) = 2T(1) + 2 = 2 + 2 = 4$.

$$T(2) \leq c2\lg 2 = 2c$$

for any $c \geq 2$.

Are we done?

Substitution method

Consider $n = 3$:

$$T(3) = 2T(1) + 3$$

But we haven't proved the $n = 1$ case!

Substitution method

We can handle this as a second base case:

$$T(3) = 2T(1) + 3 = 5.$$

Then

$$T(3) \leq c2\lg 3 \approx 3.17c$$

Again, $c \geq 2$ suffices.

Substitution method

Sometimes this is a little tricky...

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

This is similar to this recurrence:

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

So we could guess that this is also O(n).

Substitution method

Attempt 1 with induction:

Assume it's $T(m) \leq cm$ for all $m < n$. Then

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= cn + 1 \end{aligned}$$

But this is not $\leq cn$ for any $c > 0$

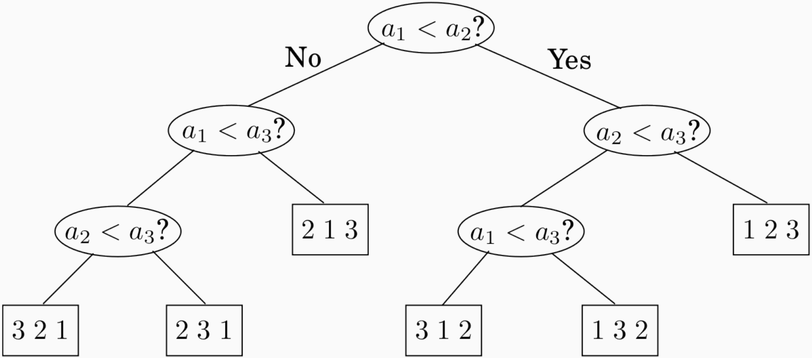
Substitution method

Attempt 2: guess $T(n) \leq cn - d$ instead.

Assume it's $T(m) \leq cm - d$ for all $m < n$. Then

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor - d + c\lceil n/2 \rceil - d + 1 \\ &= cn - 2d + 1 \\ &\leq cn - d \end{aligned}$$

Lower bound for sorting



Lower bound for sorting

- Every permutation must be considered
- There are $n!$ permutations
- so tree must have at least $n!$ leaves

Lower bound for sorting

- A tree of depth d has $\leq 2^d$ leaves
- So we must have a depth of at least $\log(n!)$

Lower bound for sorting

And $\log(n!) \geq cn \log n$ for some $c > 0$

Note that $n! \geq (n/2)^{n/2}$ since $n! = 1 \cdot 2 \cdot \cdots \cdot n$ has at least $n/2$ factors larger than $n/2$

$$\begin{aligned}
\log(n!) &\geq \log((n/2)^{n/2}) \\
&= (n/2) \log(n/2) \\
&= (n/2)(\log n - \log 2) \\
&= 1/2 n \log n - (n/2) \log 2 \\
&\geq 1/2 n \log n
\end{aligned}$$

Lower bound for sorting

Thus we must make $\Omega(n \log n)$ comparisons to reach a leaf.

So any sorting algorithm must take $\Omega(n \log n)$ time.

Then merge sort is optimal, with $\Theta(n \log n)$ time.

Quicksort

- Divide: partition A[p..r] into A[p..q-1] and A[q+1..r] such that the first is all $\leq$ A[q] and the second is all $\geq$ A[q]
- Conquer: recursively sort the two subarrays

The value A[q] is called the pivot.

Quicksort

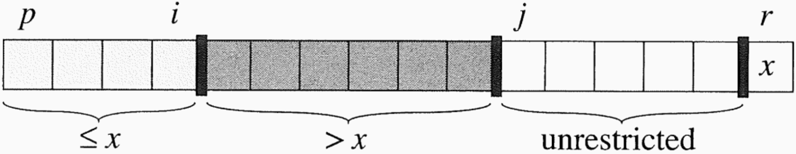
```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

Quicksort

PARTITION(A, p, r)

```
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Quicksort

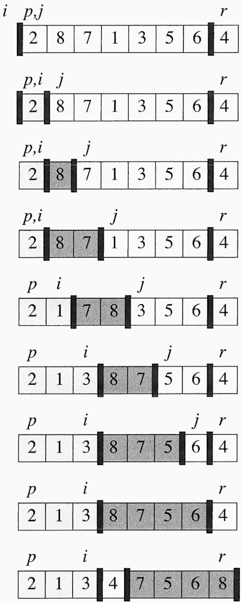


Quicksort

Suppose we have the following array:

2	8	7	1	3	5	6	4

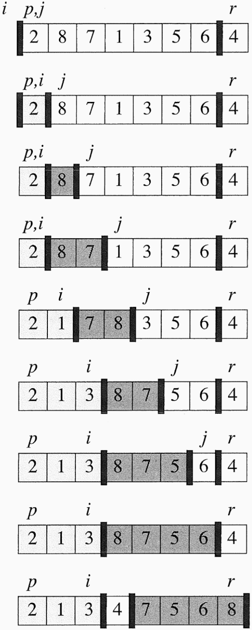
Quicksort



i p,j r
2 8 7 1 3 5 6 4
p,i j r
2 8 7 1 3 5 6 4
p,i j r
2 8 7 1 3 5 6 4
p,i j r
2 8 7 1 3 5 6 4
p i j r
2 1 7 8 3 5 6 4
p i j r
2 1 3 8 7 5 6 4
p i j r
2 1 3 8 7 5 6 4
p i r
2 1 3 8 7 5 6 4
p i r
2 1 3 4 7 5 6 8

Quicksort

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



Quicksort

What is the running time?

- How long does partitioning take?
- What's the worst case?
- Best case?

Quicksort

What is the running time (worst case)?

- Partitioning gives a maximally unbalanced set of regions
- Since we use one element as the pivot, $n - 1$ elements remain unsorted

$T(n) = T(n-1) + O(n)$

$T(n) = O(n^2)$

Quicksort

What is the running time (best case)?

- Partitioning gives a balanced set of regions
- Each side has roughly $n/2$ elements

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

$T(n) = O(n \log n)$