Recurrences

David Kauchak
cs140
Spring 2024

1

Administrative

How was assignment 0?

Mentor hours posted

Group assignment: must attend mentor hours on Thursday or Friday and submit group assignment

Assignment 1 (due Sunday): must work with **different** partner

2

Big O: Upper bound

O(g(n)) is the set of functions:

$$O(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

3

Proving bounds: find constants that satisfy inequalities

Show that $5n^2 - 15n + 100$ is $\Theta(n^2)$

Step 1: Prove $O(n^2)$ – Find constants c and n_0 such that $5n^2 - 15n + 100 \le cn^2$ for all $n > n_0$

$$cn^2 \ge 5n^2 - 15n + 100$$

$$c \ge 5 - 15/n + 100/n^2$$

Let $n_0 = 1$ and $c = 5 + 100 = 105$.
$100/n^2$ only gets smaller as n increases and we ignore $-15/n$ since it only varies between -15 and 0

4

Proving bounds

Step 2: Prove $\Omega(n^2)$ – Find constants c and n_0 such that $5n^2 - 15n + 100 \geq cn^2$ for all n > n_0

$$cn^2 \leq 5n^2 - 15n + 100$$
$$c \leq 5 - 15/n + 100/n^2$$

Let n_0 =4 and c = 5 – 15/4 = 1.25 (or anything less than 1.25). -15/n is always increasing and we ignore $100/n^2$ since it is always between 0 and 100.

5

Bounds

Is $5n^2$ $O(n)$? **No**

How would we prove it?

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

6

Disproving bounds

Is $5n^2$ $O(n)$?

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Assume it's true.

That means there exists some c and n_0 such that

$$5n^2 \leq cn \text{ for } n > n_0$$
$$5n \leq c$$ contradiction!

7

Divide and Conquer

Divide: Break the problem into smaller sub-problems

Conquer: Solve the sub-problems. Generally, this involves waiting for the problem to be small enough that it is trivial to solve (i.e. 1 or 2 items)

Combine: Given the results of the solved sub-problems, combine them to generate a solution for the complete problem

8

Divide and Conquer: some thoughts

Often, the sub-problem is the same as the original problem

Dividing the problem in half frequently does the job

May have to get creative about how the data is split

Splitting tends to generate run times with log *n* in them

9

Divide and conquer

One approach:

- Pretend like you have a working version of your function, but it only works on smaller sub-problems
- If you split up the current problem in some way (e.g. in half) and solved those sub-problems, how could you then get the solution to the larger problem?

10

MergeSort

```
MERGE-SORT(A)
if length[A] == 1
        return A
else
        q ← ⌊length[A] /2⌋
        create arrays L[1..q] and R[q + 1.. length[A]]
        copy A[1..q] to L
        copy A[q + 1.. length[A]] to R
        LS ← MERGE-SORT(L)
        RS ← MERGE-SORT(R)
        return MERGE(LS, RS)
```

12

MergeSort: Merge

Assuming L and R are sorted already, merge the two to create a single sorted array

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
        if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
                B[k] ← L[i]
                i ← i + 1
        else
                B[k] ← R[j]
                j ← j + 1
return B
```

13

Merge

L: 1 3 5 8　　　R: 2 4 6 7

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
        if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
                B[k] ← L[i]
                i ← i + 1
        else
                B[k] ← R[j]
                j ← j + 1
return B
```

14

Merge

L: 1 3 5 8　　　R: 2 4 6 7

B:

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
        if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
                B[k] ← L[i]
                i ← i + 1
        else
                B[k] ← R[j]
                j ← j + 1
return B
```

15

Merge

i　　　j

L: 1 3 5 8　　　R: 2 4 6 7

B:

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
        if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
                B[k] ← L[i]
                i ← i + 1
        else
                B[k] ← R[j]
                j ← j + 1
return B
```

16

Merge

i　　　j

L: 1 3 5 8　　　R: 2 4 6 7

B:

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
        if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
                B[k] ← L[i]
                i ← i + 1
        else
                B[k] ← R[j]
                j ← j + 1
return B
```

17

Merge

L: 1 3 5 8   R: 2 4 6 7

B: 1

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
    if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
        B[k] ← L[i]
        i ← i + 1
    else
        B[k] ← R[j]
        j ← j + 1
return B
```

18

Merge

L: 1 3 5 8   R: 2 4 6 7

B: 1

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
    if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
        B[k] ← L[i]
        i ← i + 1
    else
        B[k] ← R[j]
        j ← j + 1
return B
```

19

Merge

L: 1 3 5 8   R: 2 4 6 7

B: 1 2

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
    if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
        B[k] ← L[i]
        i ← i + 1
    else
        B[k] ← R[j]
        j ← j + 1
return B
```

20

Merge

L: 1 3 5 8   R: 2 4 6 7

B: 1 2

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
    if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
        B[k] ← L[i]
        i ← i + 1
    else
        B[k] ← R[j]
        j ← j + 1
return B
```

21

Merge

i: 3, j: 4

L: 1 3 5 8   R: 2 4 6 7

B: 1 2 3

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
        if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
                B[k] ← L[i]
                i ← i + 1
        else
                B[k] ← R[j]
                j ← j + 1
return B
```

22

Merge

i: 5, j: 4

L: 1 3 5 8   R: 2 4 6 7

B: 1 2 3

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
        if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
                B[k] ← L[i]
                i ← i + 1
        else
                B[k] ← R[j]
                j ← j + 1
return B
```

23

Merge

i: 5, j: 6

L: 1 3 5 8   R: 2 4 6 7

B: 1 2 3 4

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
        if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
                B[k] ← L[i]
                i ← i + 1
        else
                B[k] ← R[j]
                j ← j + 1
return B
```

24

Merge

i: 5, j: 6

L: 1 3 5 8   R: 2 4 6 7

B: 1 2 3 4

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
        if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
                B[k] ← L[i]
                i ← i + 1
        else
                B[k] ← R[j]
                j ← j + 1
return B
```

25

Merge

i ↓ j ↓

L: 1 3 5 8   R: 2 4 6 7

B: 1 2 3 4 5

```
Merge(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
      if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
            B[k] ← L[i]
            i ← i + 1
      else
            B[k] ← R[j]
            j ← j + 1
return B
```

26

Merge

i ↓ j ↓

L: 1 3 5 8   R: 2 4 6 7

B: 1 2 3 4 5

```
Merge(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
      if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
            B[k] ← L[i]
            i ← i + 1
      else
            B[k] ← R[j]
            j ← j + 1
return B
```

27

Merge

i ↓ j ↓

L: 1 3 5 8   R: 2 4 6 7

B: 1 2 3 4 5 6

```
Merge(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
      if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
            B[k] ← L[i]
            i ← i + 1
      else
            B[k] ← R[j]
            j ← j + 1
return B
```

28

Merge

i ↓ j ↓

L: 1 3 5 8   R: 2 4 6 7

B: 1 2 3 4 5 6

```
Merge(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
      if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
            B[k] ← L[i]
            i ← i + 1
      else
            B[k] ← R[j]
            j ← j + 1
return B
```

29

Merge

i j

L: 1 3 5 8 R: 2 4 6 7

B: 1 2 3 4 5 6 7

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
        if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
                B[k] ← L[i]
                i ← i + 1
        else
                B[k] ← R[j]
                j ← j + 1
return B
```

30

Merge

i j

L: 1 3 5 8 R: 2 4 6 7

B: 1 2 3 4 5 6 7

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
        if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
                B[k] ← L[i]
                i ← i + 1
        else
                B[k] ← R[j]
                j ← j + 1
return B
```

31

Merge

i j

L: 1 3 5 8 R: 2 4 6 7

B: 1 2 3 4 5 6 7 8

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
        if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
                B[k] ← L[i]
                i ← i + 1
        else
                B[k] ← R[j]
                j ← j + 1
return B
```

32

Merge

Running time?

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
        if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
                B[k] ← L[i]
                i ← i + 1
        else
                B[k] ← R[j]
                j ← j + 1
return B
```

36

Merge

Running time? Θ(n) - linear

```
MERGE(L, R)
create array B of length length[L] + length[R]
i ← 1
j ← 1
for k ← 1 to length[B]
        if j > length[R] or (i ≤ length[L] and L[i] ≤ R[j])
                B[k] ← L[i]
                i ← i + 1
        else
                B[k] ← R[j]
                j ← j + 1
return B
```

37

MergeSort

Running time?

```
MERGE-SORT(A)
if length[A] == 1
        return A
else
        q ← ⌊length[A] /2⌋
        create arrays L[1..q] and R[q + 1.. length[A]]
        copy A[1..q] to L
        copy A[q + 1.. length[A]] to R
        LS ← MERGE-SORT(L)
        RS ← MERGE-SORT(R)
        return MERGE(LS, RS)
```

38

Merge-Sort

Running time?

$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + D(n) + C(n) & \text{otherwise} \end{cases}$$

D(n): cost of splitting (dividing) the data

C(n): cost of merging/combining the data

39

Merge-Sort

Running time?

$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + D(n) + C(n) & \text{otherwise} \end{cases}$$

D(n): cost of splitting (dividing) the data - linear Θ(n)

C(n): cost of merging/combining the data – linear Θ(n)

40

Merge-Sort

Running time?

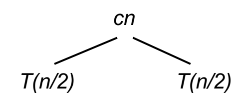
$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

Which is?

41

Merge-Sort

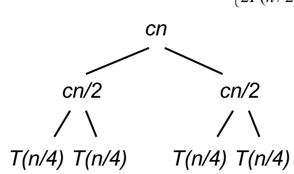
$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$



42

Merge-Sort

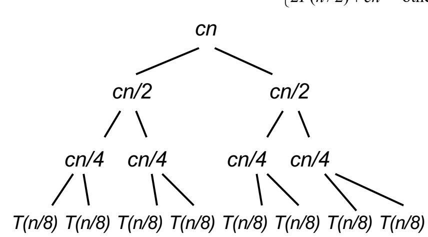
$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$



43

Merge-Sort

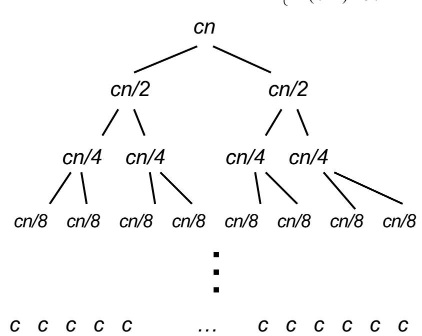
$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$



44

Merge-Sort

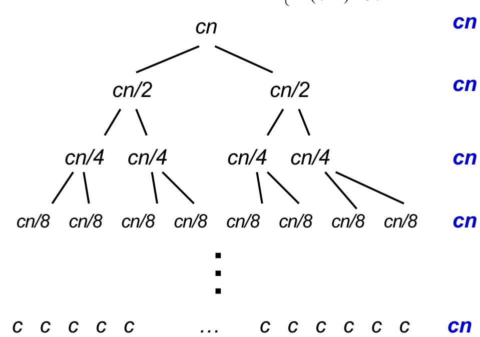
$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$



45

Merge-Sort

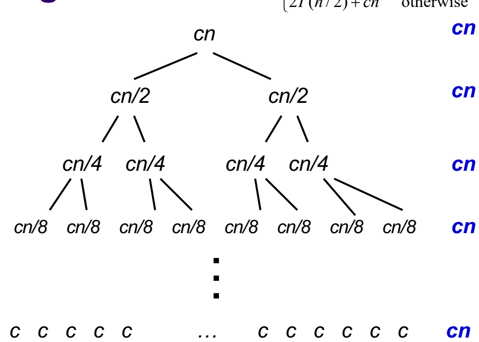
$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$



46

Merge-Sort

$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$



Depth?

47

Merge-Sort

We can calculate the depth, by determining when the recursion gets to down to a small problem size, e.g. 1

At each level, we divide by 2

$$\frac{n}{2^d} = 1$$

$$2^d = n$$

$$\log 2^d = \log n$$

$$d \log 2 = \log n$$

$$d = \log_2 n$$

48

Merge-Sort

$$T(n) = \begin{cases} c & \text{if } n \text{ is small} \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

Running time?

- Each level costs *cn*
- log *n* levels

cn log *n* = Θ(*n* log *n*)

Why don't we write it as n $\log_2$ n?

49

Log properties

$$\log_a b = \frac{\log b}{\log a}$$

$$n \log_2 n = \frac{n \log n}{\log 2}$$

$$n \log_2 n = \frac{n \log n}{c} = \theta(n \log n)$$

50

Recurrence

A function that is defined with respect to itself on smaller inputs

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

$$T(n) = 16T(n/4) + n$$

$$T(n) = 2T(n-1) + n^2$$

51

Why are we interested in recurrences?

Computational cost of divide and conquer algorithms

$$T(n) = aT(n/b) + D(n) + C(n)$$

- *a* subproblems of size *n/b*
- *D(n)* the cost of dividing the data
- *C(n)* the cost of recombining the subproblem solutions

In general, the runtimes of most recursive algorithms can be expressed as recurrences

52

The challenge

Recurrences are often easy to define because they mimic the structure of the program

But… they do not directly express the computational cost, i.e. n, n^2, …

We want to remove self-recurrence and find a more understandable form for the function

53

Three approaches

Substitution method: when you have a good guess of the solution, prove that it's correct

Recursion-tree method: If you don't have a good guess, the recursion tree can help

- Calculate exactly (like we did with MergeSort)
- Use it to get a good quest, then prove with substitution method.

Master method: Provides solutions for recurrences of the form:

$$T(n) = aT(n/b) + f(n)$$

54

Substitution method

Guess the form of the solution

Then prove it's correct by induction

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

Halves the input then a constant amount of work

Guesses?

55

Substitution method

Guess the form of the solution

Then prove it's correct by induction

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

Halves the input then a constant amount of work

Similar to binary search:

Guess: O(log n)

56

Proof?

$$T(n) = T(n/2) + d = O(\log n)$$

Ideas?

57

Proof?

$$T(n) = T(n/2) + d = O(\log n)$$

Proof by induction!
-Assume it's true for smaller T(k), i.e. k < n
-prove that it's then true for current T(n)

58

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

Assume *T(k) = O(log k)* for all *k < n*
Show that *T(n) = O(log n)*

From our assumption, *T(n/2) = O(log n/2):*

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

From the definition of big-*O*: *T(n/2) ≤ c log(n/2)*

How do we now prove T(n) = O(log n)?

59

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

To prove that *T(n) = O(log n)* identify the appropriate constants:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

i.e. some constant *c'* such that *T(n) ≤ c' log n*

$$\begin{aligned} \mathrm{T}(n) &= T(n/2) + d \\ &\le c \log\left(\frac{n}{2}\right) + d \quad \text{from our inductive hypothesis} \\ &\le c \log n - c \log 2 + d \\ &\le c \log n - c + d \quad \text{residual} \end{aligned}$$

Key question: does a constant exist such that:
$T(n) \le c' \log n$

60

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

To prove that *T(n) = O(log n)* identify the appropriate constants:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

i.e. some constant *c'* such that *T(n) ≤ c' log n*

Key question: does a constant exist such that:
$T(n) \le c' \log n$

$$\mathrm{T(n)} \le c \log n - c + d$$

if $c \ge d$, then, just let c' = c

$$\mathrm{T(n)} \le c \log n - c + d \le c \log n$$

61

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

To prove that *T(n) = O(log n)* identify the appropriate constants:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

i.e. some constant *c'* such that *T(n) ≤ c' log n*

Key question: does a constant exist such that:
$T(n) \le c' \log n$

$$\mathrm{T(n)} \le c \log n - c + d$$

if $c < d$, let c' = d+1 and

$$\mathrm{T(n)} \le c \log n - c + d \le d \log n + \log n$$

62

Base case?

For an inductive proof we need to show two things:

- *Show that it holds for some base case*
- Assuming it's true for *k* < *n* show it's true for *n*

What is the base case in our situation?

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

63

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

Guess the solution?

At each iteration, does a linear amount of work (i.e. iterate over the data) and reduces the size by one at each step

$O(n^2)$

Assume $T(k) = O(k^2)$ for all $k < n$

- again, this implies that $T(n-1) \le c(n-1)^2$

Show that $T(n) = O(n^2)$, i.e. $T(n) \le c'n^2$

64

$$T(n) = T(n-1) + n$$
$$\leq c(n-1)^2 + n \quad \text{from our inductive hypothesis}$$
$$= c(n^2 - 2n + 1) + n$$
$$= cn^2 - 2cn + c + n \quad \text{residual}$$

if $-2cn + c + n \leq 0$

then let c' = c and there exists a constant such that $T(n) \leq c'n^2$

65

$$T(n) = T(n-1) + n$$
$$\leq c(n-1)^2 + n \quad \text{from our inductive hypothesis}$$
$$= c(n^2 - 2n + 1) + n$$
$$= cn^2 - 2cn + c + n \quad \text{residual}$$

$$-2cn + c + n \leq 0$$
$$-2cn + c \leq -n$$
$$c(-2n + 1) \leq -n$$
$$c \geq \frac{n}{2n-1}$$
$$c \geq \frac{1}{2 - 1/n}$$

which holds for any $c \geq 1$ for $n \geq 1$

66

Substitution method

Guess the form of the solution

Then prove it's correct by induction

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

Halves the input then constant amount of work

Similar to binary search:

Guess: $O(\log_2 n)$

67

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

Guess the solution?

Recurses into 2 sub-problems that are half the size and performs some operation on all the elements

$O(n \log n)$

What if we guess wrong, e.g. $O(n^2)$?

Assume $T(k) = O(k^2)$ for all $k < n$

- again, this implies that $T(n/2) \leq c(n/2)^2$

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

68

$$\begin{aligned} T(n) &= 2T(n/2)+n \\ &\le 2c(n/2)^2+n \\ &= 2cn^2/4+n \\ &= 1/2cn^2+n \\ &= cn^2-(1/2cn^2-n) \end{aligned}$$

from our inductive hypothesis

residual

if

$$-(1/2cn^2-n)\le 0$$
$$-1/2cn^2+n\le 0$$
$$cn\ge 2$$

overkill?

69

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

What if we guess wrong, e.g. O(n)?

Assume $T(k) = O(k)$ for all $k < n$

- again, this implies that $T(n/2) \le c(n/2)$

Show that $T(n) = O(n)$

$$\begin{aligned} T(n) &= 2T(n/2)+n \\ &\le 2cn/2+n \\ &= cn+n \\ &\le cn \end{aligned}$$

factor of n so we can just roll it in?

70

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

What if we guess wrong, e.g. O(n)?

Assume $T(k) = O(k)$ for all $k < n$

- again, this implies that $T(n/2) \le c(n/2)$

Show that $T(n) = O(n)$

$$\begin{aligned} T(n) &= 2T(n/2)+n \\ &\le 2cn/2+n \\ &= cn+n \\ &\le cn \end{aligned}$$

Must prove the exact form!

$cn+n \le cn$??

~~factor of n so we can just roll it in?~~

71

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

Prove $T(n) = O(n \log_2 n)$

Assume $T(k) = O(k \log_2 k)$ for all $k < n$

- again, this implies that $T(k) = ck \log_2 k$

Show that $T(n) = O(n \log_2 n)$

$$\begin{aligned} T(n) &= 2T(n/2)+n \\ &\le 2cn/2\log(n/2)+n \\ &\le cn(\log_2 n-\log_2 2)+n \\ &\le cn\log_2 n-cn+n \\ &\le cn\log_2 n \end{aligned}$$

residual

if $cn \ge n$, $c > 1$

72