Data Structures & Algorithms

Divide and Conquer

- Mergesort
- Maximum Sum Subarray
- Integer Multiplication
- Guess and Induction

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 usage.

- 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 in linear time.

Consequence.

- Brute force: n^2.
- Divide-and-conquer: $n \log n$.

Mergesort

Sorting

Sorting. Given n elements, rearrange in ascending order.

Applications.

- Sort a list of names.
- Organize an MP3 library.
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Find duplicates in a mailing list.
- . . .

Mergesort

Mergesort.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

A	L	G	O	R	I	T	H	M	S			
A	L	G	O	R		I	T	H	M	S	divide	$O(1)$
A	G	L	O	R		H	I	M	S	T	sort	$2T(n/2)$
A	G	H	I	L	M	O	R	S	T		merge	$O(n)$

Merging

Merging. Combine two pre-sorted lists into a sorted whole.

How to merge efficiently?

- Linear number of comparisons.
- Use temporary array.

A	G	L	O	R		H	I	M	S	T

A	G	H	I						

```
MergeSort(A[1..n])
If n = 1 then return A
L = MergeSort(A[1..n/2])
R = MergeSort(A[n/2+1..n])
i=1, j=1
for k = 1 to n do
   if (i ≤ n/2 and j ≤ n/2) then
      if (L[i] ≤ R[j]) then B[k]=L[i], i=i+1
      else B[k]= R[j], j=j+1
   else if (i > n/2) then B[k]= R[j], j=j+1
   else B[k]=L[i], i=i+1
return B
```

A Useful Recurrence Relation

Def. T(n) = number of comparisons to mergesort an input of size n.

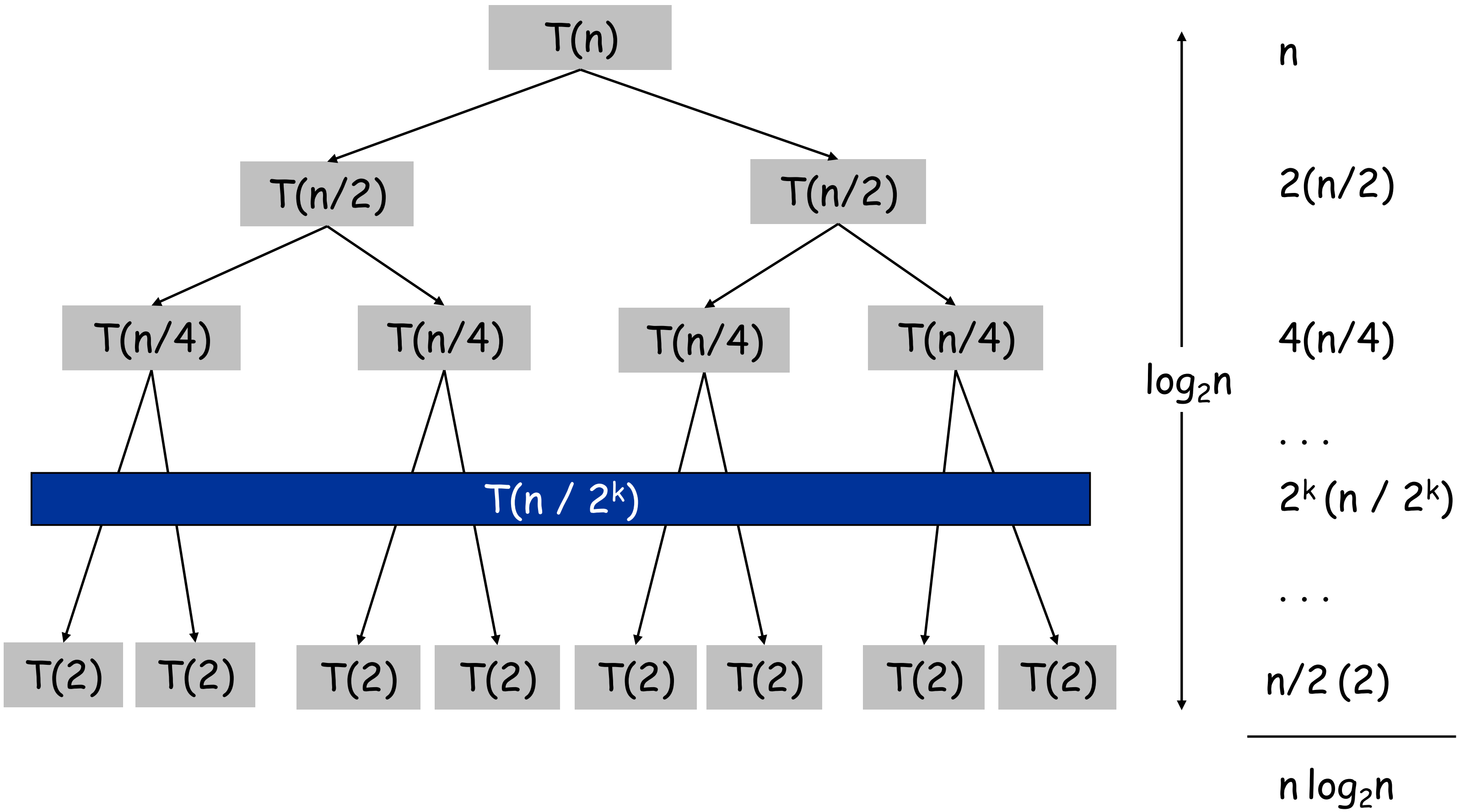
Mergesort recurrence.

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Solution. $T(n) = O(n \log_2 n)$.

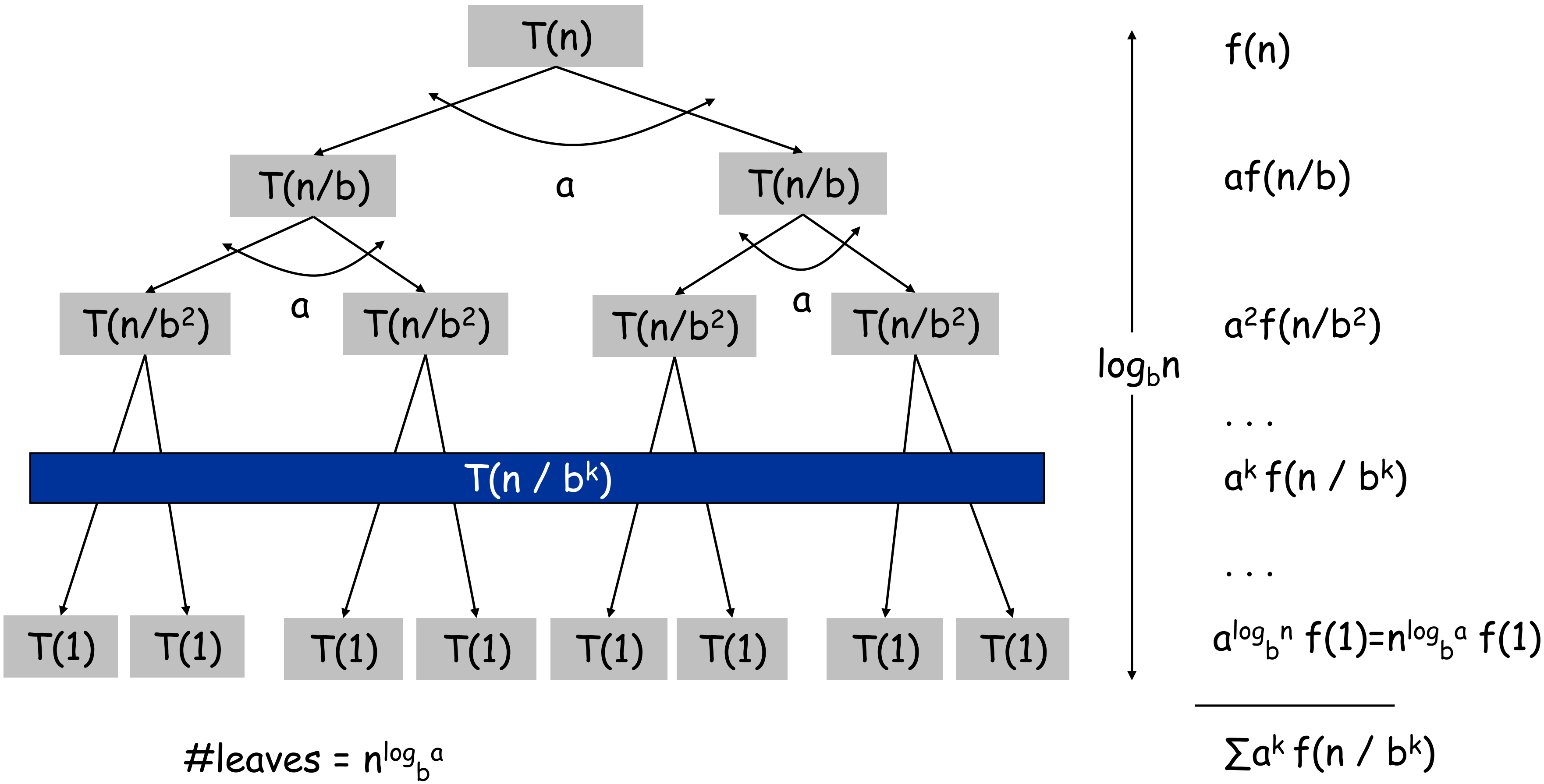
Proof by Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{aT(n/b)}_{\text{sub-problems}} + \underbrace{f(n)}_{\text{merging}} & \text{otherwise} \end{cases}$$



Master Theorem

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{aT(n/b)}_{\text{sub-problems}} + \underbrace{f(n)}_{\text{merging}} & \text{otherwise} \end{cases}$$

You can imagine above as a recursive function which calls itself: a times, each with an input of size n/b, and merge their outputs in f(n) time.

Fighting between #leaves and f(n)

- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
- If $f(n)$ polynomially greater than $n^{\log_b a}$, then $T(n)=\Theta(f(n))$
- If $n^{\log_b a}$ polynomially greater than $f(n)$, then $T(n)=\Theta(n^{\log_b a})$

g(n) is polynomially greater than h(n) iff $g(n)/h(n) = \Omega(n^x)$ for some $x > 0$

Note. The total input injecting to sub-problems is (a/b)n. Then if a/b is smaller, your running time is better.

The master theorem

Special case: f(n) = $O(n^d)$

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

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

Examples

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

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

- $T(n) = 4\,T(n/2) + O(n)$
- $T(n) = O(n^2)$

$a = 4$
$b = 2$
$d = 1$

$\log_b a > d$

- $T(n) = 3\,T(n/2) + O(n)$
- $T(n) = O(n^{\log\_2(3)} \approx n^{1.6})$

$a = 3$
$b = 2$
$d = 1$

$\log_b a > d$

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

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

$\log_b a = d$

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

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

$\log_b a < d$

Mergesort

What happen if we divide the array into more subproblems?.

- We have to find the minimum among a numbers in the merging step.
- So,

$$T(n)=aT(n/a)+an$$

- It is easy to see T(n) is minimum when a=2

Maximum Sum Subarray

Maximum Sum Subarray

Problem: Given a one dimensional array A[1..n] of numbers.
Find a contiguous subarray with largest sum within A.

Assume an empty subarray has sum 0.

Example:

4	-7	12	5	-2	3	-5	1	5	-8	2	5

4	-7	12	5	-2	3	-5	1	5	-8	2	5

Algorithm (brute-force)

Observation: Let S[i] = A[1]+…+A[i]. We have A[i]+…+A[j]=S[j]-S[i-1]

```
Pre-Processing
S[0] = 0
for i = 1 to n do
    S[i] = S[i-1]+A[i]
```

Running time of pre-processing: $T(n) = O(n)$

```
sol = 0
for i = 1 to n do
   for j = i to n do
         if S[j]-S[i-1] > sol then
            sol = S[j]-S[i-1]
return sol
```

Running time: $T(n) = O(n^2)$

Algorithm (divide and conquer)

The general strategy: Divide into 2 equal-size subarrays

Case 1: optimal solution is in one subarray

Case 2: optimal solution crosses the splitting line

4	-7	12	5	-2	3	-5	1	5	-8	2	5

```
MCS(A[1..n])
if n = 1 then return max(0, a[1])
sol = max(MCS(A[1…n/2]), MCS(A[n/2+1…n])
Lsol = 0
for i = n/2 downto 1 do
    if S[n/2]-S[i-1] > Lsol then
        Lsol = S[n/2]-S[i-1]
Rsol = 0
for i = n/2+1 to n do
    if S[i]-S[n/2-1] > Rsol then
        Rsol = S[i]-S[n/2-1]
return max(sol, Lsol+Rsol)
```

Running time: $T(n) = 2T\left(\frac{n}{2}\right) + O(n) \rightarrow T(n) = O(n \log n)$

Integer Multiplication

Integer Addition

Addition. Given two n-bit integers a and b, compute $a + b$.
Grade-school. $\Theta(n)$ bit operations.

1	1	1	1	1	1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
1	0	1	0	1	0	0	1	0

Remark. Grade-school addition algorithm is optimal.

Integer Multiplication

Multiplication. Given two n-bit integers a and b, compute $a \times b$.
Grade-school. $\Theta(n^2)$ bit operations.

```
                  1 1 0 1 0 1 0 1
                × 0 1 1 1 1 1 0 1
                -----------------
                  1 1 0 1 0 1 0 1
                0 0 0 0 0 0 0 0
              1 1 0 1 0 1 0 1
            1 1 0 1 0 1 0 1
          1 1 0 1 0 1 0 1
        1 1 0 1 0 1 0 1
      1 1 0 1 0 1 0 1
    0 0 0 0 0 0 0 0
-------------------------------
0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 1
```

Q. Is grade-school multiplication algorithm optimal?

Divide-and-Conquer Multiplication: Warmup

To multiply two n-bit integers a and b:
- Multiply four $\frac{1}{2}n$-bit integers, recursively.
- Add and shift to obtain result.

$$
\begin{aligned}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= \left(2^{n/2} \cdot a_1 + a_0\right)\left(2^{n/2} \cdot b_1 + b_0\right) = 2^n \cdot a_1 b_1 + 2^{n/2} \cdot \left(a_1 b_0 + a_0 b_1\right) + a_0 b_0
\end{aligned}
$$

Ex. $\quad \boldsymbol{a} = \underbrace{\texttt{1000}}_{a_1}\underbrace{\texttt{1101}}_{a_0} \qquad \boldsymbol{b} = \underbrace{\texttt{1110}}_{b_1}\underbrace{\texttt{0001}}_{b_0}$

$$
T(n) = \underbrace{4T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \Rightarrow T(n) = \Theta(n^2)
$$

Karatsuba Multiplication

To multiply two n-bit integers a and b:

- Add two $\frac{1}{2}n$ bit integers.
- Multiply three $\frac{1}{2}n$-bit integers, recursively.
- Add, subtract, and shift to obtain result.

$$
\begin{aligned}
a &= 2^{n/2} \cdot a_1 \ + \ a_0 \\
b &= 2^{n/2} \cdot b_1 \ + \ b_0 \\
ab &= 2^{n} \cdot a_1 b_1 \ + \ 2^{n/2} \cdot \left(a_1 b_0 + a_0 b_1\right) + a_0 b_0 \\
&= 2^{n} \cdot \underset{1}{a_1 b_1} \ + \ 2^{n/2} \cdot \left(\underset{2}{(a_1 + a_0)(b_1 + b_0)} - \underset{1}{a_1 b_1} - \underset{3}{a_0 b_0}\right) + \underset{3}{a_0 b_0}
\end{aligned}
$$

Karatsuba Multiplication

To multiply two n-bit integers a and b:

- Add two $\frac{1}{2}n$ bit integers.
- Multiply three $\frac{1}{2}n$-bit integers, recursively.
- Add, subtract, and shift to obtain result.

$$
\begin{aligned}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= 2^n \cdot a_1 b_1 + 2^{n/2} \cdot \left(a_1 b_0 + a_0 b_1\right) + a_0 b_0 \\
&= \underset{1}{2^n \cdot a_1 b_1} + 2^{n/2} \cdot \left(\underset{2}{(a_1 + a_0)(b_1 + b_0)} - \underset{1}{a_1 b_1} - \underset{3}{a_0 b_0}\right) + \underset{3}{a_0 b_0}
\end{aligned}
$$

$$
T(n) \le \underbrace{T\left(\lfloor n/2 \rfloor\right) + T\left(\lceil n/2 \rceil\right) + T\left(1 + \lceil n/2 \rceil\right)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, subtract, shift}} \Rightarrow T(n) = O(n^{\lg 3}) = O(n^{1.585})
$$

Guess and Induction

Guess and Induction

Step 1: Generate a guess at the correct answer.

Step 2: Try to prove that your guess is correct.

(Step 3: Cleanup)

First Example

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \text{ with } T(0) = 0, T(1) = 1.$$

The Master Method says $T(n) = O(n \log(n))$.

We will prove this via Guess and Induction.

Step 1: Guess the answer

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$$

Expand $T\left(\frac{n}{2}\right)$

$$T(n) = 2 \cdot \left(2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$$

Simplify

$$T(n) = 4 \cdot T\left(\frac{n}{4}\right) + 2n$$

Expand $T\left(\frac{n}{4}\right)$

$$T(n) = 4 \cdot \left(2 \cdot T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n$$

Simplify

$$T(n) = 8 \cdot T\left(\frac{n}{8}\right) + 3n$$

...

Guessing the pattern: $T(n) = 2^t \cdot T\left(\frac{n}{2^t}\right) + t \cdot n$

Plug in $t = \log(n)$, and get

$$T(n) = n \cdot T(1) + \log(n) \cdot n = n(\log(n) + 1)$$

Step 2: Prove the guess is correct.

Inductive Hypothesis: $T(n) = n(\log(n) + 1)$.

Base Case (n=1): $T(1) = 1 = 1 \cdot (\log(1) + 1)$

Inductive Step:

- Assume Inductive Hyp. for $1 \leq n < k$:
  - Suppose that $T(n) = n(\log(n) + 1)$ for all $1 \leq n < k$.
- Prove Inductive Hyp. for n=k:
  - $T(k) = 2 \cdot T\left(\frac{k}{2}\right) + k$ by definition
  - $T(k) = 2 \cdot \left(\frac{k}{2}\left(\log\left(\frac{k}{2}\right) + 1\right)\right) + k$ by induction.
  - $T(k) = k(\log(k) + 1)$ by simplifying.
  - So Inductive Hyp. holds for n=k.

Conclusion: For all $n \geq 1$, $T(n) = n(\log(n) + 1)$

Step 3: Cleanup

Pretend like you never did Step 1, and just write down:

Theorem: $T(n) = O(n\log(n))$
Proof: [Whatever you wrote in Step 2]

Second Example

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 32 \cdot n$$
$$T(2) = 2$$

Step 1: Guess: $O(n \log(n))$

But I don't have such a precise guess about the form for the $O(n \log(n))$ ...

- That is, what's the leading constant?

Can I still do Step 2?

What's wrong with this?

Inductive Hypothesis: $T(n) = O(n \log(n))$

Base case: $T(2) = 2 = O(1) = O(2 \log(2))$

Inductive Step:

- Suppose that $T(n) = O(n \log(n))$ for n < k.
- Then $T(k) = 2 \cdot T\left(\frac{k}{2}\right) + 32 \cdot k$ by definition
- So $T(k) = 2 \cdot O\left(\frac{k}{2} \log\left(\frac{k}{2}\right)\right) + 32 \cdot k$ by induction
- But that's $T(k) = O(k \log(k))$, so the I.H. holds for n=k.

Conclusion:

- By induction, $T(n) = O(n \log(n))$ for all n.

What's wrong with this?

We can use the same reasoning to prove that T(n) = O(n), which is not true!

Inductive Hypothesis: $T(n) = O(n)$

Base case: $T(2) = 2 = O(1) = O(2)$

Inductive Step:

- Suppose that $T(n) = O(n)$ for n < k.
- Then $T(k) = 2 \cdot T\left(\frac{k}{2}\right) + 32 \cdot k$ by definition
- So $T(k) = 2 \cdot O\left(\frac{k}{2}\right) + 32 \cdot k$ by induction
- But that's $T(k) = O(k)$, so the I.H. holds for n=k.

Conclusion:

- By induction, $T(n) = O(n)$ for all n.

The problem is that this doesn't make any sense. We can't have "T(n) = O(n) for n < k," since the def. of big-Oh needs to hold for all n.

Second example

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 32 \cdot n$$
$$T(2) = 2$$

Step 1: Guess: $O(n \log(n))$

But I don't have such a precise guess about the form for the $O(n \log(n))$ …

- That is, what's the leading constant?

Can I still do Step 2?

Step 2: Prove it, working backwards to figure out the constant

Guess: $T(n) \leq C \cdot n \log(n)$ for some constant C.

Inductive Hypothesis (for $n \geq 2$) : $T(n) \leq C \cdot n \log(n)$

Base case: $T(2) = 2 \leq C \cdot 2 \log(2)$ as long as $C \geq 1$

Inductive Step:

Inductive step

Inductive Hypothesis: $T(n) \leq C \cdot n\log(n)$

Assume that the inductive hypothesis holds for n<k.

$$T(k) = 2T\left(\frac{k}{2}\right) + 32k$$

$$\leq 2C\frac{k}{2}\log\left(\frac{k}{2}\right) + 32k$$

$$= k(C \cdot \log(k) + 32 \ - C)$$

$\leq k(C \cdot \log(k))$ as long as $C \geq 32$.

Then the inductive hypothesis holds for n=k.

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 32 \cdot n$$

$$T(2) = 2$$

Step 3: Cleanup

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

Proof:

- Inductive Hypothesis: $T(n) \leq 32 \cdot n\log(n)$
- Base case: $T(2) = 2 \leq 32 \cdot 2\log(2)$ is true.
- Inductive step:
  - Assume Inductive Hyp. for n<k.
  - $T(k) = 2T\left(\frac{k}{2}\right) + 32k$
  - $\leq 2 \cdot 32 \cdot \frac{k}{2}\log\left(\frac{k}{2}\right) + 32k$ By the def. of T(k)
  - $= k(32 \cdot \log(k) + 32 - 32)$ By induction
  - $= 32 \cdot k\log(k)$
  - This establishes inductive hyp. for n=k.
- Conclusion: $T(n) \leq 32 \cdot n\log(n)$ for all $n \geq 2$.
  - By the definition of big-Oh, with $n_0 = 2$ and $c = 32$, this implies that $T(n) = O(n\log(n))$

Third Example

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n \text{ for } n > 10.$$

Base case: $T(n) = 1$ when $1 \leq n \leq 10$

Step 1: Let's guess $O(n)$ and try to prove it.

Step 2: prove our guess is right

$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n$ for $n > 10$.

Base case: $T(n) = 1 \text{ when } 1 \leq n \leq 10$

We don't know what C should be yet! Let's go through the proof leaving it as "C" and then figure out what works…

Inductive Hypothesis: $T(n) \leq Cn$

Base case: $1 = T(n) \leq Cn$ for all $1 \leq \mathrm{n} \leq 10$

Whatever we choose C to be, it should have C≥1

Inductive step:

- Let k > 10. Assume that the IH holds for all n so that $1 \leq n < k$.
- $T(k) \leq k + T\left(\frac{k}{5}\right) + T\left(\frac{7k}{10}\right)$

$$\leq k + C \cdot \left(\frac{k}{5}\right) + C \cdot \left(\frac{7k}{10}\right)$$

$$= k + \frac{C}{5}k + \frac{7C}{10}k$$

$$\leq Ck \text{ ??}$$

Let's solve for C and make this true!
C = 10 works.
(write out)

- (want to show that IH holds for n=k).

Conclusion:

- There is some C so that for all $n \geq 1, T(n) \leq Cn$
- By the definition of big-Oh, T(n) = O(n).

Step 3: Cleanup

$T(n) \leq n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right)$ for $n > 10$.

Base case: $T(n) = 1$ when $1 \leq n \leq 10$

(pretend we knew this all along).

Theorem: $T(n) = O(n)$

Proof:

Inductive Hypothesis: $T(n) \leq \mathbf{10}n$.

Base case: $1 = T(n) \leq \mathbf{10}n$ for all $1 \leq \mathrm{n} \leq 10$

Inductive step:

- Let k > 10. Assume that the IH holds for all n so that $1 \leq n < k$.
- $T(k) \leq k + T\left(\frac{k}{5}\right) + T\left(\frac{7k}{10}\right)$

$$\leq k + \mathbf{10} \cdot \left(\frac{k}{5}\right) + \mathbf{10} \cdot \left(\frac{7k}{10}\right)$$
$$= k + 2k + 7k = \mathbf{10}k$$

- Thus, IH holds for n=k.

Conclusion:

- For all $n \geq 1, T(n) \leq \mathbf{10}n$
- Then, T(n) = O(n), using the definition of big-Oh with $n_0 = 1, c = 10$.

Incorrect Guess

$$T(n) \leq T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n \text{ for } n > 3.$$

Base case: $T(n) = 1 \text{ when } 1 \leq n \leq 3$

Step 1: Let's guess $O(n)$ and try to prove it.

Step 2: Try to prove the guess

Inductive Hypothesis: $T(n) \leq Cn$

Base case: $1 = T(n) \leq Cn$ for all $1 \leq \mathrm{n} \leq 3$

Inductive step:

- Let k > 10. Assume that the IH holds for all n so that $1 \leq n < k$.
- $T(k) \leq k + T\left(\frac{k}{3}\right) + T\left(\frac{2k}{3}\right)$

$$\leq k + C \cdot \left(\frac{k}{3}\right) + C \cdot \left(\frac{2k}{3}\right)$$

$$= k + \frac{C}{3}k + \frac{2C}{3}k$$

$$\leq Ck \text{ ??}$$

Whatever we choose C to be, it should have C≥1

No C exists to make this true!

Conclusion:

- Our guess can not be proved by the above induction proof
- Either our guess is incorrect or we have to change our proof

Theorem

$$T(n) \le T(an) + T(bn) + O(n)$$

$$T(n) = \begin{cases} O(n) & \text{if } a + b < 1 \\ O(n \log n) & \text{if } a + b = 1 \end{cases}$$

Prove by Induction

Last Example

$$T(n) \leq 2T\left(\frac{n}{2}\right) + 1 \text{ for } n > 2,$$

$$\text{Base case: } T(1) = 1$$

Inductive Hypothesis: $T(n) \leq Cn$

Base case: $1 = T(1) \leq C(1)$ (C must be at least 1)

Inductive step:

- Let k > 1. Assume that the IH holds for all n so that $1 \leq n < k$.
- $T(k) \leq 2T\left(\frac{k}{2}\right) + 1 \leq 2C \cdot \left(\frac{k}{2}\right) + 1 \leq C$k+1 $\leq Ck$??

No C exists to make this true!

Conclusion:

- Our guess can not be proved by the above induction proof
- Either our guess is incorrect or we have to change our proof

Last Example

$$T(n) \leq 2T\left(\frac{n}{2}\right) + 1 \text{ for } n > 2,$$
$$\text{Base case: } T(1) = 1$$

Inductive Hypothesis: $T(n) \leq \boldsymbol{C}n + \mathbf{B}$

Base case: $1 = T(1) \leq \boldsymbol{C(1) + B}$

Inductive step:

- Let k > 1. Assume that the IH holds for all n so that $1 \leq n < k$.
- $T(k) \leq 2T\left(\frac{k}{2}\right) + 1 \leq 2\left(\boldsymbol{C} \cdot \left(\frac{k}{2}\right)\right) + \boldsymbol{B}) + 1 \leq \boldsymbol{Ck} + 2\boldsymbol{B} + 1 \leq \boldsymbol{C}k + \boldsymbol{B}$??

$\mathrm{B} \leq \mathbf{-1}$

The new guess

$$T(n) \leq 2n - \mathbf{1}$$

Note: You canot prove $T(n) \leq 2n$ by induction but you can prove $T(n) \leq 2n - \mathbf{1}$!!!

References

References

- Sections 5.1, 5.2, 5.4, and 5.5 of the text book "algorithm design" by Jon Kleinberg and Eva Tardos
- Section 4.1 of the text book "introduction to algorithms" by CLRS, 3rd edition.
- The original slides were prepared by Kevin Wayne. The slides are distributed by Pearson Addison-Wesley.