Divide and Conquer

- Mergesort
- Integer Multiplication
- Maximum Sum Subarray

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.

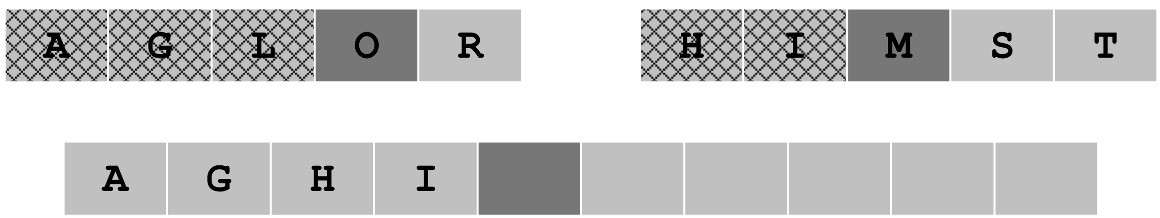
A L G O R I T H M S		
A L G O R I T H M S	divide	$O(1)$
A G L O R H I M S T	sort	$2T(n/2)$
A G H I L M O R S T	merge	$O(n)$

Merging

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

How to merge efficiently?

- Linear number of comparisons.
- Use temporary array.



Challenging version. In-place merge

using only a constant amount of extra storage

A Useful Recurrence Relation

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

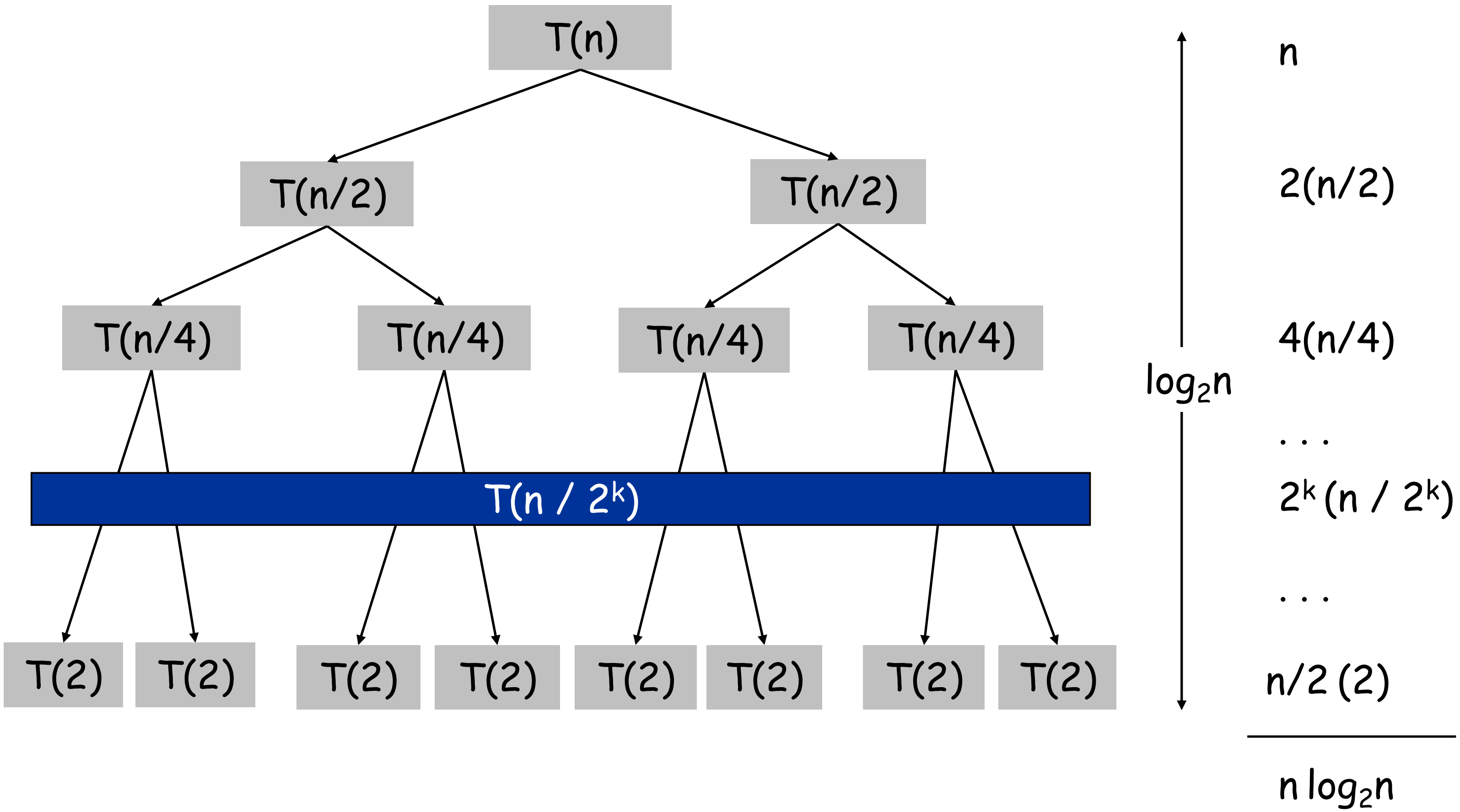
Mergesort recurrence.

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

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

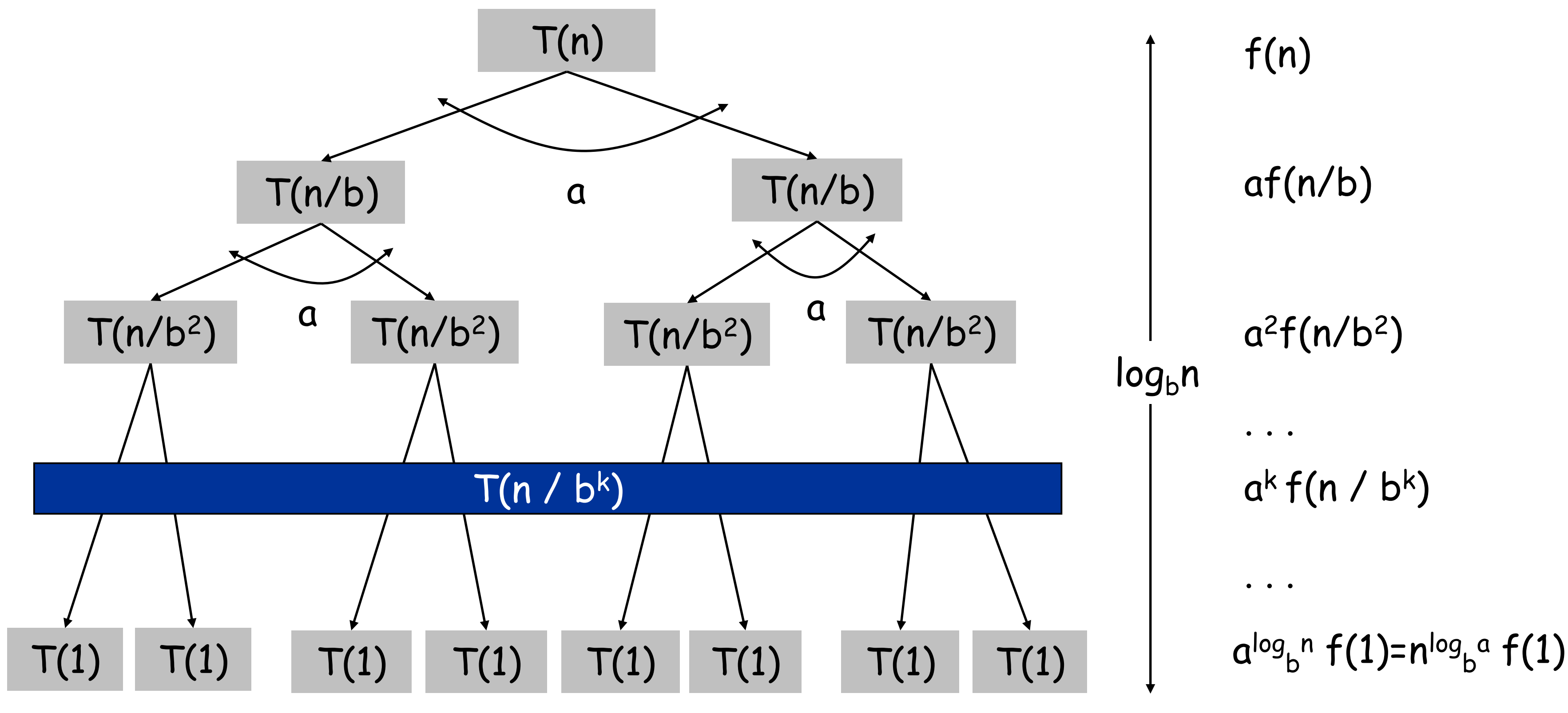
Proof by Recursion Tree

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



Recursion Tree

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



#leaves = $n^{\log_b a}$

$\sum a^k f(n / b^k)$

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 $T(n) = \Theta(n^{\log_b a})$

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

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

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 0
            1 1 0 1 0 1 0 1 0
          1 1 0 1 0 1 0 1 0
        1 1 0 1 0 1 0 1 0
      1 1 0 1 0 1 0 1 0
    1 1 0 1 0 1 0 1 0
  0 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.

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

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

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

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.

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

1 2 1 3 3

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.

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

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

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] > solR then
        Rsol = S[i]-S[n/2-1]
return max(sol, Lsol+Rsol)
```

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

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.