# Divide and Conquer

Andreas Klappenecker
[based on slides by Prof. Welch]

# Divide and Conquer Paradigm

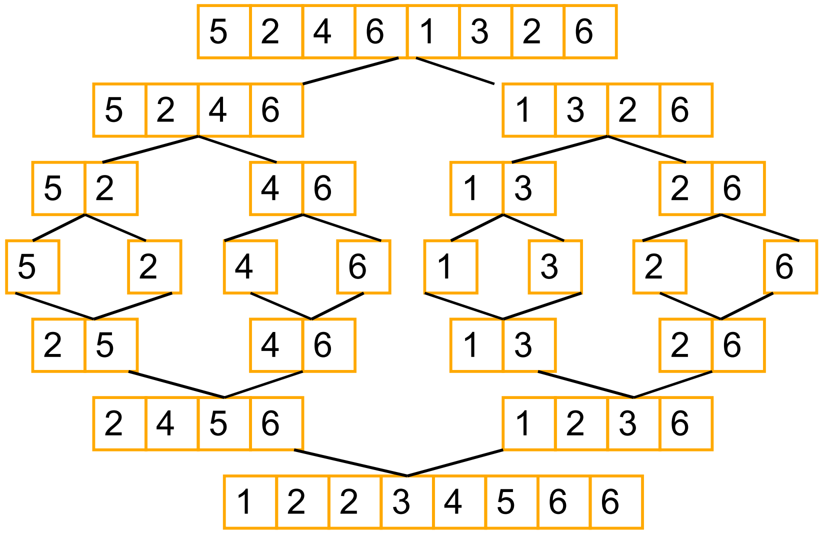
- An important general technique for designing algorithms:
  - divide problem into subproblems
  - recursively solve subproblems
  - combine solutions to subproblems to get solution to original problem
- Use recurrences to analyze the running time of such algorithms

# Mergesort

# Example: Mergesort

- DIVIDE the input sequence in half
- RECURSIVELY sort the two halves
  - basis of the recursion is sequence with 1 key
- COMBINE the two sorted subsequences by merging them

# Mergesort Example



# Mergesort Animation

- http://ccl.northwestern.edu/netlogo/models/run.cgi?MergeSort.862.378

# Recurrence Relation for

- Let $T(n)$ be worst case time on a sequence of $n$ keys
- If $n = 1$, then $T(n) = \Theta(1)$ (constant)
- If $n > 1$, then $T(n) = 2\ T(n/2) + \Theta(n)$
  - two subproblems of size $n/2$ each that are solved recursively
  - $\Theta(n)$ time to do the merge

# Recurrence Relations

# How To Solve Recurrences

- Ad hoc method:
  - expand several times
  - guess the pattern
  - can verify with proof by induction
- Master theorem
  - general formula that works if recurrence has the form $T(n) = aT(n/b) + f(n)$
    - $a$ is number of subproblems
    - $n/b$ is size of each subproblem
    - $f(n)$ is cost of non-recursive part

# Master Theorem

Consider a recurrence of the form

$T(n) = a\, T(n/b) + f(n)$

with a>=1, b>1, and f(n) eventually positive.

a) If $f(n) = O(n^{\log_b(a)-\varepsilon})$, then $T(n)=\Theta(n^{\log_b(a)})$.

b) If $f(n) = \Theta(n^{\log_b(a)})$, then $T(n)=\Theta(n^{\log_b(a)} \log(n))$.

c) If $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ and f(n) is regular, then $T(n) =\Theta(f(n))$

[f(n) regular iff eventually af(n/b)<= cf(n) for some constant c<1]

# Excuse me, what did it say???

Essentially, the Master theorem compares the function f(n)

with the function $g(n)=n^{\log_b(a)}$.

Roughly, the theorem says:

a) If $f(n) \ll g(n)$ then $T(n)=\Theta(g(n))$.

b) If $f(n) \approx g(n)$ then $T(n)=\Theta(g(n)\log(n))$.

c) If $f(n) \gg g(n)$ then $T(n)=\Theta(f(n))$.

# Déjà vu: Master Theorem

Consider a recurrence of the form

$T(n) = a\,T(n/b) + f(n)$

with a>=1, b>1, and f(n) eventually positive.

a) If $f(n) = O(n^{\log_b(a)-\varepsilon})$, then $T(n)=\Theta(n^{\log_b(a)})$.

b) If $f(n) = \Theta(n^{\log_b(a)})$, then $T(n)=\Theta(n^{\log_b(a)}\log(n))$.

c) If $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ and f(n) is regular, then $T(n) = \Theta(f(n))$

[f(n) regular iff eventually af(n/b)<= cf(n) for some constant c<1]

# Nothing is perfect…

The Master theorem does not cover all possible cases. For example, if

$$f(n) = \Theta(n^{\log_b(a)} \log n),$$

then we lie between cases 2) and 3), but the theorem does not apply.

There exist better versions of the Master theorem that cover more cases, but these are even harder to memorize.

# Idea of the Proof

Let us iteratively substitute the recurrence:

$$
\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a(aT(n/b^2)) + f(n/b)) + bn \\
&= a^2T(n/b^2) + af(n/b) + f(n) \\
&= a^3T(n/b^3) + a^2 f(n/b^2) + af(n/b) + f(n) \\
&= \ldots \\
&= a^{\log_b n}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i) \\
&= n^{\log_b a}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i)
\end{aligned}
$$

# Idea of the Proof

Thus, we obtained

$T(n) = n^{\log_b(a)}\, T(1) + \Sigma\, a^i\, f(n/b^i)$

The proof proceeds by distinguishing three cases:

1) The first term in dominant: $f(n) = O(n^{\log_b(a)-\varepsilon})$
2) Each part of the summation is equally dominant: $f(n) = \Theta(n^{\log_b(a)})$
3) The summation can be bounded by a geometric series: $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ and the regularity of f is key to make the argument work.

# Further Divide and Conquer Examples

# Additional D&C Algorithms

- binary search
  - divide sequence into two halves by comparing search key to midpoint
  - recursively search in one of the two halves
  - combine step is empty
- quicksort
  - divide sequence into two parts by comparing pivot to each key
  - recursively sort the two parts
  - combine step is empty

# Additional D&C applications

- computational geometry
  - finding closest pair of points
  - finding convex hull
- mathematical calculations
  - converting binary to decimal
  - integer multiplication
  - matrix multiplication
  - matrix inversion
  - Fast Fourier Transform

# Strassen's Matrix Multiplication

# Matrix Multiplication

- Consider two n by n matrices A and B
- Definition of AxB is n by n matrix C whose (i,j)-th entry is computed like this:
  - consider row i of A and column j of B
  - multiply together the first entries of the rown and column, the second entries, etc.
  - then add up all the products
- Number of scalar operations (multiplies and adds) in straightforward algorithm is $O(n^3)$.
- Can we do it faster?

# Divide-and-Conquer

$$A \quad \times \quad B \quad = \quad C$$

$$\begin{array}{|c|c|}\hline A_0 & A_1 \\ \hline A_2 & A_3 \\ \hline\end{array} \times \begin{array}{|c|c|}\hline B_0 & B_1 \\ \hline B_2 & B_3 \\ \hline\end{array} = \begin{array}{|c|c|}\hline A_0\times B_0+A_1\times B_2 & A_0\times B_1+A_1\times \\ \hline A_2\times B_0+A_3\times B_2 & A_2\times B_1+A_3\times \\ \hline\end{array}$$

- Divide matrices A and B into four submatrices each
- We have 8 smaller matrix multiplications and 4 additions. Is it faster?

# Divide-and-Conquer

Let us investigate this recursive version of the matrix multiplication.

Since we divide A, B and C into 4 submatrices each, we can compute the resulting matrix C by

- 8 matrix multiplications on the submatrices of A and B,
- plus $\Theta(n^2)$ scalar operations

# Divide-and-Conquer

- Running time of recursive version of straightfoward algorithm is
  - $T(n) = 8T(n/2) + \Theta(n^2)$
  - $T(2) = \Theta(1)$

  where $T(n)$ is running time on an n x n matrix
- Master theorem gives us:

  $T(n) = \Theta(n^3)$
- Can we do fewer recursive calls (fewer multiplications of the n/2 x n/2 submatrices)?

# Strassen's Matrix Multiplication

$$A \times B = C$$

$$\begin{bmatrix} A_0 & A_1 \\ A_2 & A_3 \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 \\ B_2 & B_3 \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$P_2 = (A_{21} + A_{22}) * B_{11}$$
$$P_3 = A_{11} * (B_{12} - B_{22})$$
$$P_4 = A_{22} * (B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{12}) * B_{22}$$
$$P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$$
$$P_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$$

$$C_{11} = P_1 + P_4 - P_5 + P_7$$
$$C_{12} = P_3 + P_5$$
$$C_{21} = P_2 + P_4$$
$$C_{22} = P_1 + P_3 - P_2 + P_6$$

# Strassen's Matrix Multiplication

- Strassen found a way to get all the required information with only 7 matrix multiplications, instead of 8.

- Recurrence for new algorithm is
  - $T(n) = 7T(n/2) + \Theta(n^2)$

## Solving the Recurrence Relation

Applying the Master Theorem to

$T(n) = a\,T(n/b) + f(n)$

with a=7, b=2, and $f(n)=\Theta(n^2)$.

Since $f(n) = O(n^{\log_b(a)-\varepsilon}) = O(n^{\log_2(7)-\varepsilon})$,

case a) applies and we get

$T(n)= \Theta(n^{\log_b(a)}) = \Theta(n^{\log_2(7)}) = O(n^{2.81})$.

# Discussion of Strassen's

- Not always practical
  - constant factor is larger than for naïve method
  - specially designed methods are better on sparse matrices
  - issues of numerical (in)stability
  - recursion uses lots of space
- Not the fastest known method
  - Fastest known is $O(n^{2.376})$
  - Best known lower bound is $\Omega(n^2)$