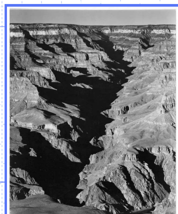
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Divide-and-Conquer

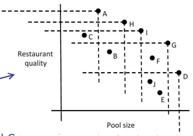


Grand Canyon from South Rim, 1941. Ansel Adams. U.S. government image. U.S. National Archives and Records Administration.

Application: Maxima Sets

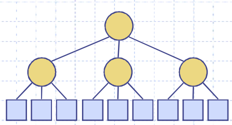
- We can visualize the various trade-offs for optimizing two-dimensional data, such as points representing hotels according to their pool size and restaurant quality, by plotting each as a two-dimensional point, (x, y), where x is the pool size and y is the restaurant quality score.
- We say that such a point is a **maximum point** in a set if there is no other point, (x′, y′), in that set such that x ≤ x′ and y ≤ y′.
- The maximum points are the best potential choices based on these two dimensions and finding all of them is the **maxima set** problem.

We can efficiently find all the maxima points by divide-and-conquer. Here the set is {A,H,I,G,D}.



Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two or more disjoint subsets S_1, S_2, …
  - Conquer: solve the subproblems recursively
  - Combine: combine the solutions for S_1, S_2, …, into a solution for S
- The base case for the recursion are subproblems of constant size
- Analysis can be done using recurrence equations

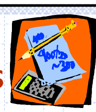


Merge-Sort Review

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Conquer: recursively sort S_1 and S_2
  - Combine: merge S_1 and S_2 into a unique sorted sequence

Algorithm *mergeSort*(S)
  Input sequence S with n elements
  Output sequence S sorted according to C
  if $S.size() > 1$
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    $mergeSort(S_1)$
    $mergeSort(S_2)$
    $S \leftarrow merge(S_1, S_2)$

Recurrence Equation Analysis

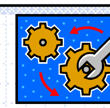


- The conquer step of merge-sort consists of merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes at most bn steps, for some constant b.
- Likewise, the basis case ($n < 2$) will take at b most steps.
- Therefore, if we let $T(n)$ denote the running time of merge-sort:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

- We can therefore analyze the running time of merge-sort by finding a closed form solution to the above equation.
  - That is, a solution that has $T(n)$ only on the left-hand side.

Iterative Substitution

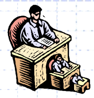


- In the iterative substitution, or “plug-and-chug,” technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$\begin{aligned} T(n) &= 2T(n/2) + bn \\ &= 2(2T(n/2^2)) + b(n/2)) + bn \\ &= 2^2T(n/2^2) + 2bn \\ &= 2^3T(n/2^3) + 3bn \\ &= 2^4T(n/2^4) + 4bn \\ &= \ldots \\ &= 2^iT(n/2^i) + ibn \end{aligned}$$

- Note that base, T(n)=b, case occurs when 2^i=n. That is, i = log n.
- So, $T(n) = bn + bn\log n$
- Thus, T(n) is O(n log n).

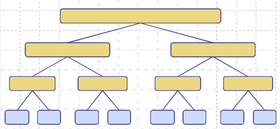
The Recursion Tree



- Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \ge 2 \end{cases}$$

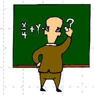
depth	T's	size	time
0	1	n	bn
1	2	$n/2$	bn
i	2^i	$n/2^i$	bn
…	…	…	…



Total time = $bn + bn \log n$

(last level plus all previous levels)

Guess-and-Test Method



- In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

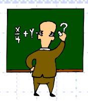
$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \ge 2 \end{cases}$$

- Guess: T(n) < cn log n.

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

- Wrong: we cannot make this last line be less than cn log n

Guess-and-Test Method, (cont.)



- Recall the recurrence equation:

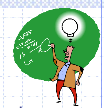
$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \ge 2 \end{cases}$$

- Guess #2: T(n) < cn $\log^2$ n.

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

  - if c > b.
- So, T(n) is O(n $\log^2$ n).
- In general, to use this method, you need to have a good guess and you need to be good at induction proofs.

Master Method



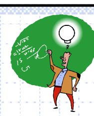
- Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$$

- The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

Master Method, Example 1



- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$
- The Master Theorem:

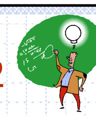
1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

- Example:

$$T(n) = 4T(n/2) + n$$

Solution: $\log_b a = 2$, so case 1 says T(n) is O(n^2).

Master Method, Example 2



- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$
- The Master Theorem:

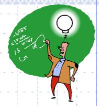
1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

- Example:

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

Solution: $\log_b a = 1$, so case 2 says T(n) is O(n $\log^2$ n).

Master Method, Example 3

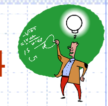


- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = T(n/3) + n \log n$$

Solution: $\log_b a=0$, so case 3 says T(n) is O(n log n).

Master Method, Example 4



- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = 8T(n/2) + n^2$$

Solution: $\log_b a=3$, so case 1 says T(n) is $O(n^3)$.

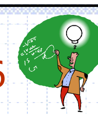
Master Method, Example 5

- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = 9T(n/3) + n^3$$

Solution: $\log_b a=2$, so case 3 says T(n) is $O(n^3)$.

Master Method, Example 6

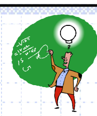


- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.
- Example:

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

Solution: $\log_b a=0$, so case 2 says T(n) is O(log n).

Master Method, Example 7

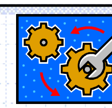


- The form: $T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = 2T(n/2) + \log n$$ (heap construction)

Solution: $\log_b a=1$, so case 1 says T(n) is O(n).

Sketch of Proof of the Master Theorem



- Using iterative substitution, let us see if we can find a pattern:

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

- We then distinguish the three cases as
  - The first term is dominant
  - Each part of the summation is equally dominant
  - The summation is a geometric series

Integer Multiplication

- Algorithm: Multiply two n-bit integers I and J.
  - Divide step: Split I and J into high-order and low-order bits

$$I = I_h 2^{n/2} + I_l$$
$$J = J_h 2^{n/2} + J_l$$

  - We can then define I*J by multiplying the parts and adding:

$$I*J = (I_h 2^{n/2} + I_l)*(J_h 2^{n/2} + J_l)$$
$$= I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l$$

  - So, T(n) = 4T(n/2) + n, which implies T(n) is O(n^2).
  - But that is no better than the algorithm we learned in grade school.

An Improved Integer Multiplication Algorithm

- Algorithm: Multiply two n-bit integers I and J.
  - Divide step: Split I and J into high-order and low-order bits

$$I = I_h 2^{n/2} + I_l$$
$$J = J_h 2^{n/2} + J_l$$

  - Observe that there is a different way to multiply parts:

$$I*J = I_h J_h 2^n + [(I_h - I_l)(J_l - J_h) + I_h J_h + I_l J_l]2^{n/2} + I_l J_l$$
$$= I_h J_h 2^n + [(I_h J_l - I_l J_l - I_h J_h + I_l J_h) + I_h J_h + I_l J_l]2^{n/2} + I_l J_l$$
$$= I_h J_h 2^n + (I_h J_l + I_l J_h)2^{n/2} + I_l J_l$$

  - So, T(n) = 3T(n/2) + n, which implies T(n) is O($n^{\log_2 3}$), by the Master Theorem.
  - Thus, T(n) is O($n^{1.585}$).

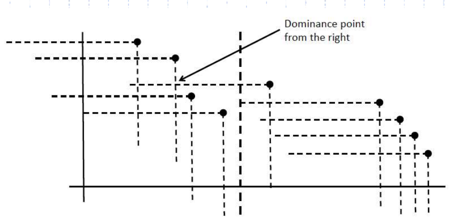
Solving the Maxima Set Problem

- Let us now return to the problem of finding a **maxima set** for a set, S, of n points in the plane.
- This problem is motivated from multi-objective optimization, where we are interested in optimizing choices that depend on multiple variables.
- For instance, in the introduction we used the example of someone wishing to optimize hotels based on the two variables of pool size and restaurant quality.
- A point is a **maximum point** in S if there is no other point, (x′, y′), in S such that x ≤ x′ and y ≤ y′.

Divide-and-Conquer Solution

- Given a set, S, of n points in the plane, there is a simple divide-and-conquer algorithm for constructing the maxima set of points in S.
- If n ≤ 1, the maxima set is just S itself.
- Otherwise, let p be the median point in S according to a lexicographic ordering of the points in S, that is, where we order based primarily on x-coordinates and then by y-coordinates if there are ties.
- Next, we recursively solve the maxima-set problem for the set of points on the left of this line and also for the points on the right.
- Given these solutions, the maxima set of points on the right are also maxima points for S.
- But some of the maxima points for the left set might be dominated by a point from the right, namely the point, q, that is leftmost.
- So then we do a scan of the left set of maxima, removing any points that are dominated by q, until reaching the point where q's dominance extends.
- The union of remaining set of maxima from the left and the maxima set from the right is the set of maxima for S.

Example for the Combine Step



Pseudo-code

Algorithm MaximaSet(S):

Input: A set, S, of n points in the plane
Output: The set, M, of maxima points in S

if $n \leq 1$ **then**
  return S
Let p be the median point in S, by lexicographic (x, y)-coordinates
Let L be the set of points lexicographically less than p in S
Let G be the set of points lexicographically greater than or equal to p in S
$M_1 \leftarrow$ MaximaSet(L)
$M_2 \leftarrow$ MaximaSet(G)
Let q be the lexicographically smallest point in M_2
for each point, r, in M_1 **do**
  if $x(r) \leq x(q)$ **and** $y(r) \leq y(q)$ **then**
    Remove r from M_1
return $M_1 \cup M_2$

A Little Implementation Detail

- Before we analyze the divide-and-conquer maxima-set algorithm, there is a little implementation detail that we need to work out.
- Namely, there is the issue of how to efficiently find the point, p, that is the median point in a lexicographical ordering of the points in S according to their (x, y)-coordinates.
- There are two immediate possibilities:
- One choice is to use a linear-time median-finding algorithm, such as that given in Section 9.2. This achieves a good asymptotic running time, but adds some implementation complexity.
- Another choice is to sort the points in S lexicographically by their (x, y)-coordinates as a preprocessing step, prior to calling the MaxmaSet algorithm on S. Given this preprocessing step, the median point is simply the point in the middle of the list.

Analysis

- In either case, the rest of the non-recursive steps can be performed in O(n) time, so this implies that, ignoring floor and ceiling functions (as allowed by the analysis of Exercise C-11.5), the running time for the divide-and-conquer maxima-set algorithm can be specified as follows (where b is a constant):

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

- Thus, according to the Master Theorem, this algorithm runs in O(n log n) time.

