CMPT 231: Data Structures and Algorithms

cmpt231.seanho.com and myCourses

11 Sep 2012
CMPT231
Dr. Sean Ho
Trinity Western University

Outline for today

- Administrivia
- Algorithms and asymptotic complexity
  - Example: Insertion sort
    - Notation: Θ, O, Ω, o, ω
- Divide-and-conquer
  - Example: Merge sort
    - Recursion and recurrence relations
  - Example: Maximum-subarray
  - Example: Matrix multiply
    - Naive method, divide-and-conquer method
    - Strassen's method

What is an algorithm?

- Well-defined process for solving a problem
  - Input → Computation → Output
- May be expressed in any appropriate language
  - Pseudocode, English, etc.
- May be implemented in many programming languages
  - Python, C, Java, etc.

- Computing science is not about toolkits (Python, C++, Java, etc.) but about problem solving

Algorithmic complexity

- Number of machine instructions needed to execute the algorithm
  - Expressed as a function of size of input
  - Constant factors are not important
- Depends on machine architecture
  - e.g., GPUs can perform many parallel operations very quickly
  - We'll ignore this in our machine model
- "Running time" (speed) is a more complex topic than just algorithmic complexity
  - Cache/memory hierarchy plays a big role

Basic machine model

- The basic instruction set we assume roughly follows most CPU architectures:
  - Arithmetic: + - * /, < > ≠, left/right bitwise shift
  - Data: load (read), store (assign), copy
  - Control: if/else, for/while, functions
  - Types: char, int, float (with fixed word size)
    - Not arbitrarily large numbers
  - Basic data structures: pointers, fixed-length arrays (*not* Python lists / STL vectors)
- Each of these basic instructions is assumed to take constant execution time

Outline for today

- Administrivia
- Algorithms and asymptotic complexity
  - Example: Insertion sort
    - Notation: Θ, O, Ω, o, ω
- Divide-and-conquer
  - Example: Merge sort
    - Recursion and recurrence relations
  - Example: Maximum-subarray
  - Example: Matrix multiply
    - Naive method, divide-and-conquer method
    - Strassen's method

Example task: sorting

- Input: array of key-value pairs
  - wlog, let keys be the integers 1 … n
  - values (payload) can be any data, staying attached to respective keys
- Output: array with elements sorted in increasing order by key
  - In-place: modify original array
  - Out-of-place: return a sorted copy
    - We'll focus on in-place sorting for now
- Standard fun: Python sort/ed(), C++/Java sort()
  - How do they do it?

Simple solution: insertion sort

Input	5	2	4	6	1	3
j=3:	2	5	4	6	1	3
j=4:	2	4	5	6	1	3
j=5:	2	4	5	6	1	3
j=6:	1	2	4	5	6	3
Out:	1	2	3	4	5	6

- e.g., a hand of cards
- insertion_sort(A, n):

```
for j = 2 to n:
    key = A[ j ]
    i = j – 1
    while i > 0 and A[ I ] > key:        # scoot over items
        A[ i + 1 ] = A[ i ]
        i = i - 1
    A[ i + 1 ] = key
```

- Loop invariant: A[1 .. j-1] are in sorted order
  - Check: before loop, during loop, after loop

Insertion sort: complexity

- Let t_j = # times the 'while' condition is checked
- insertion_sort(A, n):

```
    for j = 2 to n:                      # cost c0 * n times
        key = A[ j ]                     # c1 * n-1
        i = j – 1                        # c2 * n-1
        while i > 0 and A[ I ] > key:    # c3 * Σ_2^n t_j
            A[ i + 1 ] = A[ i ]          # c4 *  Σ_2^n (t_j - 1)
            i = i – 1                    # c5 *  Σ_2^n (t_j - 1)
        A[ i + 1 ] = key                 # c6 * n-1
```

- Summation notation: $\Sigma_2^n t_j = t_2 + t_3 + \ldots + t_n$

Insertion sort: worst-case

- Best-case is if input is already sorted:
  - Still need to scan through, but all $t_j = 1$
  - → Linear in n: can express total complexity as T(n) = $a*n + b$, for some constants a,b
- Worst case? Input in reverse-sorted order!
  - 'while' loop is always max length: $t_j = j$
  - e.g., line 5: $c4 * \Sigma_2^n (t_j - 1) = c4 * \Sigma_2^n (j - 1)$ $= c4 * (n - 1)(n)/2 = (c4 / 2) * n^2 - (½) * n$
    - Similarly for the other lines in the function
  - → Quadratic in n
- Average case: random order: $t_j = j/2$, quadratic

Θ() notation

- The constants c1, c2, … may vary on different platforms, but as n gets big, constants irrelevant
  - Even the n term gets dominated by n^2
- Insertion sort has complexity on the order of n^2
  - Notation: $T(n) = \Theta(n^2)$ ("big theta")
- $\Theta(1)$ means the algorithm runs in constant time

Outline for today

- Administrivia
- Algorithms and asymptotic complexity
  - Example: Insertion sort
    - Notation: Θ, O, Ω, o, ω
- Divide-and-conquer
  - Example: Merge sort
    - Recursion and recurrence relations
  - Example: Maximum-subarray
  - Example: Matrix multiply
    - Naive method, divide-and-conquer method
    - Strassen's method

Divide and conquer

- Insertion sort is incremental:
  - At each step, given that A[1 … j-1] is sorted, insert A[j] such that A[1 … j] is sorted
- Another design strategy:
  - Split up the task into smaller chunks
  - When chunks are small enough, solve directly (base case)
  - Combine results and return up the stack
- Can implement via function recursion or loops
- Merge sort is an example, which ends up being more efficient than insertion sort

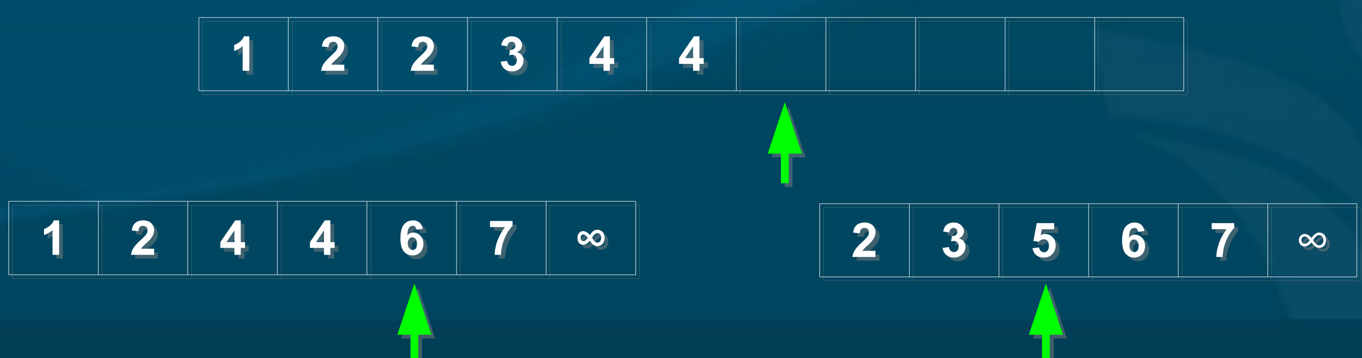
Divide and conquer: merge sort

- In English:
  - Split array in half
    - If array has only one element, we're done
  - Recurse to sort each half
  - Merge two sorted sub-arrays
- In pseudocode:

```
merge_sort(A, p, r):
    if p < r:
        q = floor( (p + r) / 2 )
        merge_sort(A, p, q)
        merge_sort(A, q+1, r)
        merge(A, p, q, r)
```

Linear-time merge

- How to do the merge?
- A[p .. q] and A[q+1 .. r] are each sorted, $p \le q < r$
- Make temp copies of each sub-array (left + right)
  - Append "infinity" item to end of each copy
- Step through both sub-array copies:
  - Compare first item from each sub-array
  - Copy smaller one into main array
    and move to next item in that list



Linear-time merge: pseudocode

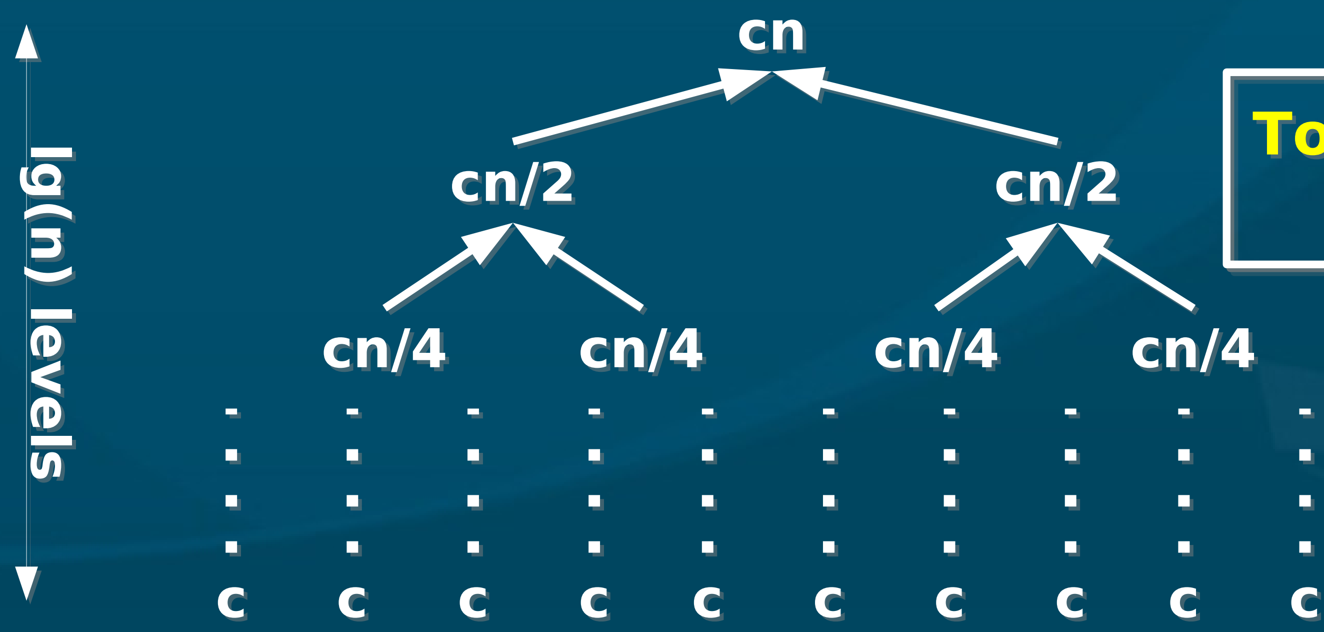
- merge(A, p, q, r):

```
    ( n1, n2 ) = ( q – p + 1, r – q )
    new arrays: L[ 1 .. n1+1 ], R[ 1 .. n2+1 ]
    for i in 1 .. n1: L[ i ] = A[ p + i – 1 ]
    for j in 1 .. n2: R[ j ] = A[ q + j ]
    ( L[ n1+1 ], R[ n2+1 ] ) = ( ∞, ∞ )
    ( i, j ) = ( 1, 1 )
    for k in p .. r:
        if L[ i ] ≤ R[ j ]:
            A[ k ] = L[ i ]
            i = i + 1
        else:
            A[ k ] = R[ j ]
            j = j + 1
```

Complexity: Θ(n)
where n = r – p + 1

Merge sort: complexity

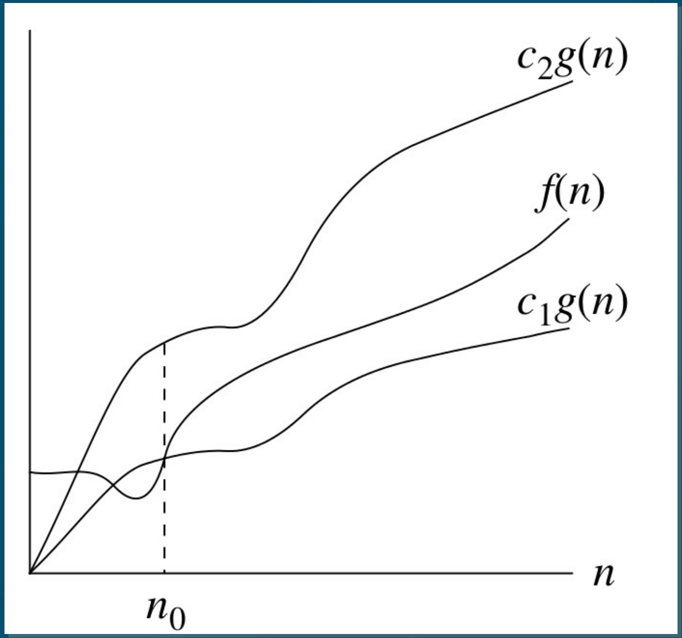
- How to analyse complexity of a recursive algo?
- Recurrence relation: base case + inductive step
- Base case: if $n = 1$, then $T(n) = \Theta(1)$
- Inductive step: if $n > 1$,
  then $T(n) = 2 * T(n/2) + \Theta(n)$



Total complexity: Θ(n lg(n))

Asymptotic growth

- Behaviour "in the limit" (for big n)
- Def: $f(n) \in \Theta(\ g(n)\ )$
  iff $\exists$ constants c_1, c_2, n_0 such that
  $0 \leq c_1\ g(n) \leq f(n) \leq c_2\ g(n)$
  for all $n > n_0$
  - $\Theta(\ g(n)\ )$ is a set of functions
  - $f(n)$ is "sandwiched" between $c_1\ g(n)$ and $c_2\ g(n)$
- "Big O": $O(\ g(n)\ )$ specifies an upper-bound
  - e.g., $\Theta(\ n^2\ ) \subset O(\ n^2\ ) \subset O(\ n^3\ )$
- "Big Omega": $\Omega(\ g(n)\ )$ specifies a lower-bound
  - Other examples?



Asymptotic short-hand

- When Θ et al. are used on the right side of =,
  - Means "there exists" $f \in \Theta(g)$
  - e.g., $2n^2 + 3n = \Theta(n^2)$
- When Θ et al. are used on the left side of =,
  - Means "for all" $f \in \Theta(g)$
  - e.g., $4n^2 + \Theta(n \lg(n)) = \Theta(n^2)$
    (this holds true for any function in $\Theta(n \lg(n))$)

Asymptotic domination

- "Little o": $f \in o(g)$ iff for all $c > 0$, there exists n_0 such that $0 \le f(n) < cg(n)$ for all $n > n_0$.
  - i.e., as $n \to \infty$, $f(n) / g(n) \to 0$
- "Little omega": $f \in \omega(g)$ iff for all $c > 0$, there exists n_0 such that $0 \le cg(n) < f(n)$ for all $n > n_0$.
  - i.e., as $n \to \infty$, $f(n) / g(n) \to \infty$
- E.g.: $n^{1.9999} = o(n^2)$, $n^2 / \lg(n) = o(n^2)$, but $n^2 / 100000 \neq o(n^2)$,
- $n^{2.000001} = \omega(n^2)$, $n^2 \lg(n) = \omega(n^2)$

Useful math identities

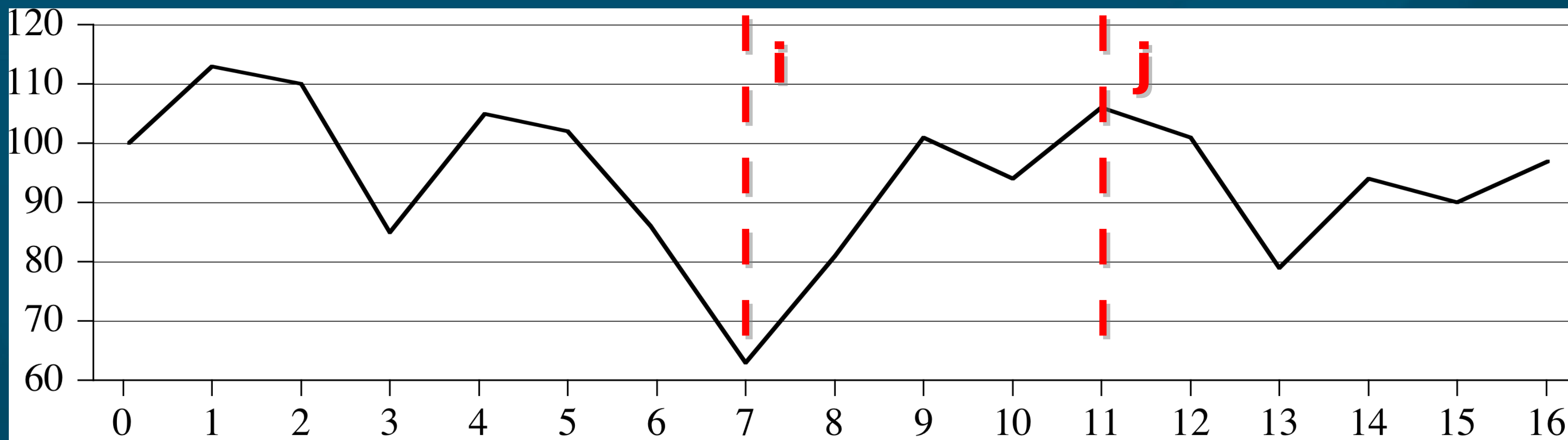
- All logs are the same up to a constant factor:
  - $\log_a(n) = \log_b(n) / \log_b(a)$
  - So we just use $\lg = \log_2$ for convenience
- $\Theta(1) \subset o(\lg(n)) \subset o(n) \subset o(n^{p>1}) \subset o(p^n)$
- In fact, for all a>1 and b: $n^b / a^n \to 0$ as $n \to \infty$.
  - Hence, $n^b = o(a^n)$
- $n! = n(n-1)(n-2)\ldots(2)(1)$
  Stirling's approximation: $$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right),$$
  - hence $\lg(n!) = \Theta(n \lg(n))$

Outline for today

- Administrivia
- Algorithms and asymptotic complexity
  - Example: Insertion sort
    - Notation: Θ, O, Ω, o, ω
- Divide-and-conquer
  - Example: Merge sort
    - Recursion and recurrence relations
  - Example: Maximum-subarray
  - Example: Matrix multiply
    - Naive method, divide-and-conquer method
    - Strassen's method

Maximum subarray

- A more complex example of divide-and-conquer
- Input: array A[1..n] of numbers (some negative)
- Output: indices (i,j) that maximize sum(A[i..j])
  - e.g., daily change in stock price:
    when was optimal time to buy (i) & sell (j)?

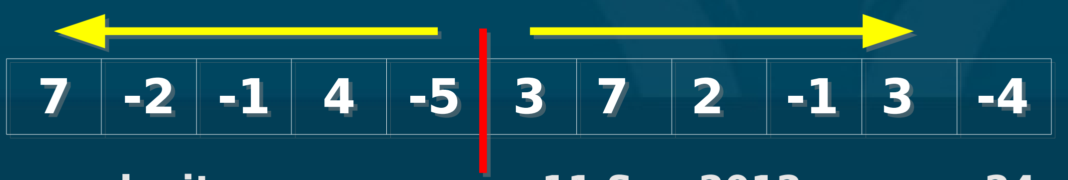


Brute force: $\theta(n^2)$

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change A[]		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Max subarray: algorithm

- Divide-and-conquer can do it in θ(n lg(n)):
  - Split array in half
  - Recursively find max subarray in each half
    - (What's the base case?)
  - Find max subarray which spans the midpoint
  - Pick the best out of the 3 subarrays and return
- Finding max subarray spanning midpoint in θ(n):
  - Decrement i from mid down to low to maximize sum(A[i .. mid])
  - Increment j from mid+1 up to high to maximize sum(A[mid+1 .. j])



Max subarray: complexity

- max_subarray(A, low, mid, high): → $T(n)$
  - Split → $\theta(1)$
  - Recurse on each half → $2T(n/2)$
  - Subarray spanning midpoint → $\theta(n)$
  - Return best of 3 → $\theta(1)$
- Recurrence relation:
  - Inductive step: $T(n) = 2T(n/2) + \theta(n)$
  - Base case: $T(1) = \theta(1)$
- Same recurrence as merge sort: $\theta(n \lg(n))$
- Actually, max subarray can be done in $\theta(n)$!
  - See exercise #4.1-5

Outline for today

- Administrivia
- Algorithms and asymptotic complexity
  - Example: Insertion sort
    - Notation: Θ, O, Ω, o, ω
- Divide-and-conquer
  - Example: Merge sort
    - Recursion and recurrence relations
  - Example: Maximum-subarray
  - Example: Matrix multiply
    - Naive method, divide-and-conquer method
    - Strassen's method

Example: matrix multiply

- Input: two n x n matrices A[i,j] and B[i,j]
- Output: n x n matrix C = A * B:
  - C[i,j] = $\Sigma^n_{k=1}$ (A[i,k] B[k,j])
- Simplest method:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

```
for i in 1 .. n:
    for j in 1 .. n:
        for k in 1 .. n:
            C[i,j] += A[i,k] * B[k,j]
```

  - Complexity? Can we do better?

Basic divide-conquer algorithm

- Divide-and-conquer: split matrices into 4 parts:
  - (assume n is a power of 2)

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

- Recurse 8 times to get products of sub-matrices
- Add and combine into result:
  - $C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$,
  - $C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$,
  - $C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$,
  - $C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$.
- Base case?
- (Generalise to when n is not a power of 2?)

Basic div-conq: complexity

- Split of matrices can be constant time if done using indices rather than copying matrices
- Each recursive call takes $T(n/2)$; do 8 of them
- Combining results takes $\Theta(n^2)$ due to addition (each entry in C[] requires one addition)
- $\Rightarrow$ Recurrence relation: $T(n) = 8T(n/2) + \Theta(n^2)$
  - Base case: $T(1) = \Theta(1)$
- Doing 8 recursive calls kills us here;
  total complexity is still $\Theta(n^3)$,
  no better than brute-force
- If we can save even 1 recursive call,
  even at the expense of $o(n^2)$ of work, it will help

Strassen's method

- Make 10 sums of submatrices: $S_1 = B_{12} - B_{22}$,
  $S_2 = A_{11} + A_{12}$, $S_3 = A_{21} + A_{22}$, $S_4 = B_{21} - B_{11}$,
  $S_5 = A_{11} + A_{22}$, $S_6 = B_{11} + B_{22}$, $S_7 = A_{12} - A_{22}$,
  $S_8 = B_{21} + B_{22}$, $S_9 = A_{11} - A_{21}$, $S_{10} = B_{11} + B_{12}$.
- Recurse 7 times to get 7 products: $P_1 = A_{11} * S_1$,
  $P_2 = S_2 * B_{22}$, $P_3 = S_3 * B_{11}$, $P_4 = A_{22} * S_4$,
  $P_5 = S_5 * S_6$, $P_6 = S_7 * S_8$, $P_7 = S_9 * S_{10}$.
- Add products and combine for result:
  $C_{11} = P_5 + P_4 - P_2 + P_6$, $C_{12} = P_1 + P_2$,
  $C_{21} = P_3 + P_4$, $C_{22} = P_5 + P_1 - P_3 - P_7$.

Strassen: complexity

- Even though more sums are done, they are all still $\Theta(n^2)$ and so don't change asymptotic cplxity
  - Although for smaller n it may not be worth it
- Recurrence: $T(n) = 7T(n/2) + \Theta(n^2)$
  - $T(1) = \Theta(1)$
- Solution to the recurrence is $T(n) = \Theta(n^{\lg 7})$
- In general, for $T(n) = a\,T(n/b) + \Theta(f(n))$:
  - if $f(n)$ is smaller than $O(n^{\log\_b(a)})$:
    - Then $T(n) = \Theta(n^{\log\_b(a)})$
    - Leaves dominate recursion tree
- One case of the "master theorem"