cs3230 notes

Tags	
Type	Notes

📌 **algorithm:** sequence of unambiguous and executable instructions for solving a problem (obtain a valid output given a valid input)

properties of good algorithms

1. correctness
2. generality: applicable to wide range of inputs
3. device independent (as far as possible)
4. efficient in terms of time, space, resources (worst/average/best case)
5. usable as "subroutine" for other problems
6. simple to code, understand, and debug
7. well documented

dealing with really large outputs

- applying modulo to results ($m \approx 2^{wordsize}$)

analysis of algorithms

- model of computation: RAM
- every instruction takes constant amount of time
- counting number of instructions needed
- complexity based on input size

running time $T(n)$

- worst case: maximum time needed for any input of size (at most) n
- average case: expected time taken over all inputs of size n
  - assumes all inputs are equally probable (or follows some probability distribution)

comparing efficiencies

- matters only for large sized inputs

asymptotic analysis

- not measuring actual run time
- for large inputs, how does the run time behave?
- often ignore constant multiplicative factors
- nothing to do with best/worst/average case runtime
  - asymptotic analysis happens within each class of runtime

steps to proof

1. find a c, n_0 that fit the definition for each of the terms of f
2. add up all your c, take the max of your n_0
3. add up all your inequalities to get the final inequality you want
4. explain what c and n_0 are

using limits to determine bounds

- assume $f(n), g(n) > 0$
- $\lim_{n\to\infty}(\frac{f(n)}{g(n)}) = 0 \Rightarrow f(n) = o(g(n))$
- $\lim_{n\to\infty}(\frac{f(n)}{g(n)}) < \infty \Rightarrow f(n) = O(g(n))$
- $0 < \lim_{n\to\infty}(\frac{f(n)}{g(n)}) < \infty \Rightarrow f(n) = \Theta(g(n))$

- $\lim_{n\to\infty}(\frac{f(n)}{g(n)}) > 0 \Rightarrow f(n) = \Omega(g(n))$
- $\lim_{n\to\infty}(\frac{f(n)}{g(n)}) = \infty \Rightarrow f(n) = \omega(g(n))$

common time complexities

- in order of increasing time complexity

1. $O(1)$
2. $O(a^n), a < 1$
3. $O(\lg \lg n)$
4. $O(\lg n)$
5. $O(n)$
6. $O(n \lg n)$
7. $O(n^k), k > 1$
8. $O(a^n), a > 1$
9. $O(n!)$

- $O(n^k) > O(n \lg n)$ but n may have to be very large if $1 < n \leq 2$

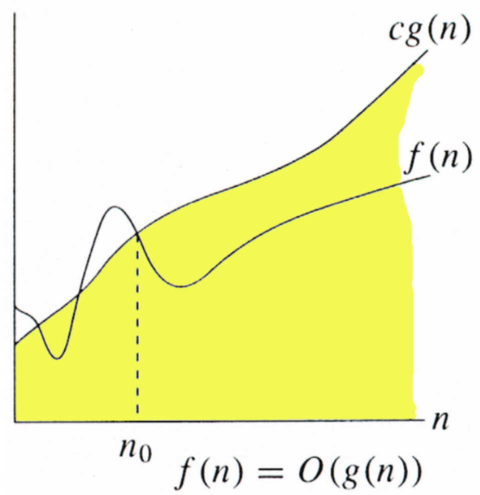
properties

- reflexivity: for O, Ω, Θ, $f(n) = O(f(n))$
- transitivity: for all, $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- symmetry: $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$
- complementarity:
  - $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$
  - $f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$

upper bound: $f \in O(g)$

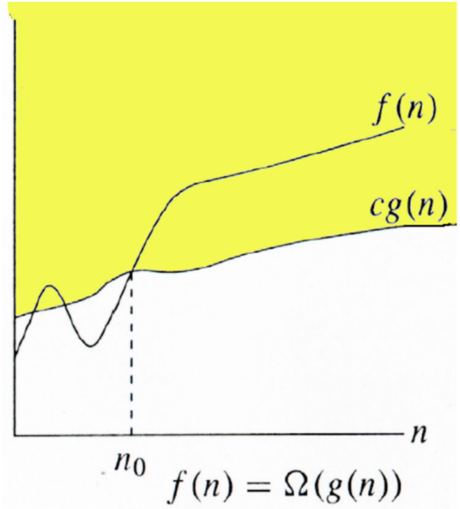
- if there exists constant $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0 : 0 \leq f(n) \leq cg(n)$
- g is an upper bound on f

- $O(g) = \{f : \exists c > 0, n_0 > 0 : \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$
- f grows no faster than g



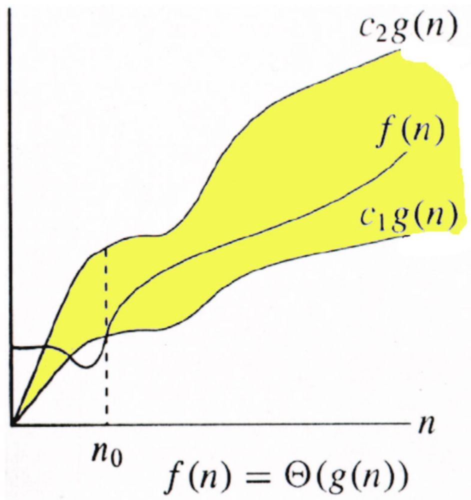
lower bound: $f \in \Omega(g)$

- if there exists a constant $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0 : 0 \leq cg(n) \leq f(n)$
- g is a lower bound on f
- $\Omega(g) = \{f : \exists c > 0, n_0 > 0 : \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$
- f grows no slower than g
  - think of it as: “nothing gets worse than what g is”
  - also can think of it as “it takes at least $\Omega(g)$ to run”
- while $f \in \Omega(1)$ is always a possibility, a better lower bound is one that for a much larger n_0 works as a “lower bound”



tight bound: $f \in \Theta(g)$

- if there exists a constant $c_1, c_2 > 0$ and $n_0 > 0$ such that $\forall n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
- g is a tight bound on f
- $\Theta(g) = \{f : \exists c_1, c_2 > 0, n_0 > 0 : \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$



- $\Theta(n) = O(g) \cap \Omega(g)$

strict upper bound: $f \in o(g)$

- if for all constant $c > 0$ there exists a constant $n_0 > 0$ such that $\forall n \geq n_0$: $0 \leq f(n) < cg(n)$
- g is a strict upper bound on f
- $o(g) = \{f : \forall c > 0, \exists n_0 > 0 : \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$

strict lower bound: $f \in \omega(g)$

- if for all constant $c > 0$ there exists a constant $n_0 > 0$ such that $\forall n \geq n_0$: $0 \leq cg(n) < f(n)$
- g is a strict lower bound on f
- $\omega(g) = \{f : \forall c > 0, \exists n_0 > 0 : \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$

recurrences

- approximating $\lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor$ to be $\frac{n}{2}$

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n), n > 1 \\ \Theta(1), n = 1 \end{cases}$$

- base case usually omitted
- often taken as constant for small (constant) size input

telescoping method

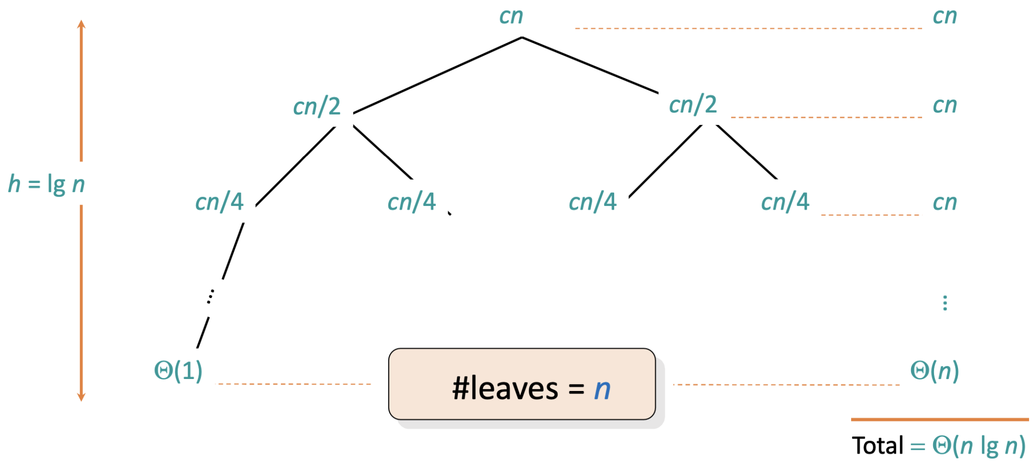
- for any sequence $a_0, a_1, \ldots, a_n$, $\sum_{k=0}^{n-1}(a_k - a_{k+1}) = a_0 - a_n$
- given $T(n) = aT(\frac{n}{b}) + f(n)$, express it as $\frac{T(n)}{g(n)} = \frac{T(\frac{n}{b})}{g(\frac{n}{b})} + h(n)$ where $h(n) = \frac{f(n)}{g(n)}$
  - think of how to divide the expression
  - ℓ is the height of the recurrence

$$
\begin{aligned}
&\frac{T(n)}{g(n)} = \frac{T(n/b)}{g(n/b)} + h(n)\\
&\frac{T(n/b)}{g(n/b)} = \frac{T(n/2b)}{g(n/2b)} + h(n)\\
&\dots\\
&\frac{T(b)}{g(b)} = \frac{T(1)}{g(1)} + h(n)
\end{aligned}
$$

$$
\begin{aligned}
&\Rightarrow \frac{T(n)}{g(n)} = \frac{T(1)}{g(1)} + \ell \times h(n)\\
&\Rightarrow T(n) = g(n) \times T(1) + \ell \times h(n) \times g(n)\\
&\Rightarrow T(n) \in O(\ell \cdot h(n) \cdot g(n))
\end{aligned}
$$

recursion tree

- given recurrence $T(n) = g(n)T(k(n)) + f(n)$, draw a recursion tree
  - calculate the depth of the tree
  - calculate the work done per level
  - total work: depth * work done per level
    - alternative: sum work per level across depth
- common $g(n)$, $k(n)$ heights:
  - $g(n) = 1, k(n) = \frac{n}{b} \Rightarrow \log n$
  - $g(n) = \sqrt{n}, k(n) = \sqrt{n} \Rightarrow \log \log n$
  - $g(n) = 1, k(n) = n - 1 \Rightarrow n$



master theorem

- given recurrence of form $T(n) = aT(n/b) + f(n)$ where $a \geq 1, b > 1$ and f is asymptotically positive
- let $c_{crit} = \log_b(a)$
- compare $f(n)$ against $n^{c_{crit}}$, focusing on the power of n^d if it exists in $f(n)$
- case 1 ($c_{crit} > d$): $f(n) = O(n^{c_{crit}-\varepsilon})$ for some constant $\varepsilon > 0$
  - work done at the leaves is more than that at the top
  - $f(n)$ grows asymptotically slower than $n^{c_{crit}}$ by a factor of n^{ε}
  - $T(n) \in \Theta(n^{\log_b(a)})$
- case 2 ($c_{crit} = d$): $f(n) = \Theta(n^{c_{crit}} \log^k(n))$ for some constant $k \geq 0$
  - work done at every level is the same
  - $f(n)$ and $n^{c_{crit}}$ grows at similar rates
  - $T(n) \in \Theta(n^{\log_b(a)} \log^{k+1}(n))$
- case 3 ($c_{crit} < d$): $f(n) = \Omega(n^{c_{crit}} + \varepsilon)$ for some constant $\varepsilon > 0$
  - work done at the root is more than that of the other levels
  - $f(n)$ grows polynomially faster than $n^{c_{crit}}$ by a factor of n^{ε}
  - $f(n)$ must also satisfy the regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$
    - guarantees that sum of subproblems is smaller than $f(n)$
  - $T(n) \in \Theta(f(n))$

substitution method

- guess the form of the solution ($O(f(n))$) and verify by induction
- for induction
  - choose values for c and n_0
  - prove base case where $n = n_0 = 1$ chosen such that $T(1)$ is satisfied
  - recursive case for $n > 1$
    - using strong induction, assume $T(k) \leq cf(n)$ for $n > k \geq 1$
    - use $T(n)$ and solve for the recurrence using induction
- choice of induction hypothesis is very important
  - key: split up the constants across $c_1, c_2, \ldots$ since they cannot be treated as the same

correctness

- on all valid inputs, the algorithm gives correct outputs
- parts of a correctness proof:
  - invariants: define invariants (every recursive call or loop)
  - initialization: prove that invariants are true at the start
  - maintainance: show (usually by induction) that if they are true at the start, invariants hold true at the start of the next iteration
  - conclusion: conclude that at the end, the algorithm gives the right answer
- iterative algorithms:
  - inner loops rely on correctness of outer loops
  - the end of loop invariant is used to prove correctness of termination
  - show that
    - invariant true at initialization
    - correctly maintained
    - implies correctness with termination condition

- recursive algorithms:
  - usually induction based on parameters of algorithm (e.g. length of search for binary search)
  - base case: same base cases as recursion where we explicitly prove those work
  - induction step (using strong induction): if all other calls of algorithm work, then the recursive call to these sub-calls will also work given all the possible cases

divide and conquer

1. divide the problem into smaller subproblems
2. solve the subproblems recursively (conquer)
3. combine/use subproblem solutions to get the solution to the full problem

$$T(n) = aT(n/b) + f(n)$$

* may not always work with master theorem

- uses induction for proof of correctness (given recursive algorithm)

sorting

- input: sequence $(a_1, a_2, \ldots, a_n)$ of comparable objects
- output: permutation $(a_1', a_2', \ldots, a_n')$ of input such that $a_i' \leq a_{i+1}'$ for $0 \leq i < n$
- properties;
  - small runtime across worst case and average case
  - simple
  - in-place sorting
  - stability
  - comparison based

in-place

- uses constant (or very little) extra memory besides the input list
- insertion sort ($O(1)$ extra space)
- randomized quick sort ($O(\log n)$ extra space)

stable

- for "equal" elements, the original ordering is preserved
- can be maintained using an auxiliary array to indicate the position of elements
- insertion and merge sort

comparison-based

- elements can only be compared with each other
- no other property of elements can be used
- insertion, merge, heap, and quick sort
- best worst-case runtime: $O(n \log n)$
- modelled using decision tree where each comparison is a node and leaf nodes is the sorted list based
  - each node is $a_i \leq a_j$
  - left subtree: yes, right subtree: no
  - worst case running time (number of comparisons performed) is longest path from root to leaf

📌 **theorem:** any comparison based algorithm takes at least $\Omega(n \log n)$ time

- model the algorithm as a tree and the tree contains at least $n!$ leaves for every possible permutation of the input
  - height of tree is at least $\log(n!) = n \log n - n \log e + O(\log n) \approx \Omega(n \log n)$ (using Stirling's approximation)

> 📌 **corollary:** merge sort is optimal for comparison based sorting

quick sort

```
def quick_sort(A, p, r):
    if p >= r: return
    pivot = A[p]
    q = partition(A, p, r, pivot)
    quick_sort(A, p, q - 1)
    quick_sort(A, q + 1, r)

quick_sort(A, 0, n - 1)
```

- worst case: array sorted $T(n) = T(j-1) + T(n-j) + O(n) \in \Theta(n^2)$
- average case: distinct and sorted and choosing an ideal pivot that provides uniform mapping $O(n \log n)$
  - choose pivot at random to ensure uniform distribution of permutations

  > 📌 **theorem:** probability that the run time of randomized quick sort exceeds average by $x\% = n^{-\frac{x}{100} \ln \ln n}$

  - probability the run time of randomized quick sort is double the average given $n \geq 10^6$ is 10^{-15}

counting sort $\Theta(n+k)$

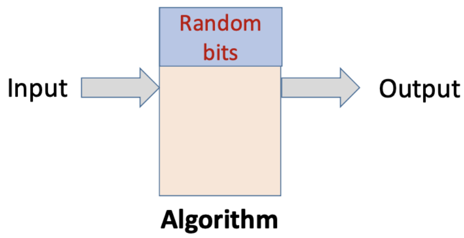
- not comparison-based sorting
- input: $A[1..n]$, where $A[i] \in \{1, 2, \ldots, k\}$
- output: $B[1..n]$ (sorted version of A)
- $C[1..k]$ holds the number of elements smaller than or equal to i
  - final sorted array is created by moving i to $B[C[i-1]+1]$ to $B[C[i]]$
- if $k = O(n)$, then $\Theta(n)$ time

- if $k \geq n \log n$, $O(n + k) \geq O(n \log n)$

radix sort $\Theta(\frac{bn}{\log n})$/$\Theta(dn)$

- sort least significant digit/bits first using counting sort since number of digits is small
- stable sorting
- b bit word broken into b/r groups of r bit words
  - r must be chosen well
  - b/r passes
  - each pass is $\Theta(n + 2^r)$
  - total: $\Theta(\frac{b}{r}(n + 2^r))$
  - optimal $r = \log n$
    - $\Theta(\frac{bn}{\log n})$
  - if numbers in range $[1, n^d]$, then $b = d \log n$: $\Theta(dn)$

randomized algorithms



- output and running time are functions of the input and random bits chosen
  - inputs cannot be controlled so randomize other things
- potential outcomes: TLE or WA
- if given a set of random bits and the answer is always right, then random bits are useless
- types:

    - las vegas: output is always correct, runtime has small probability of being too large and overall expected runtime is good
    - monte carlo: output is not always correct (but with small probability), overall runtime is consistently good
- analysis relies on linearity of expectation: $E(A+B) = E(A) + E(B)$
- increasing the probability of success involves increasing the number of times the experiment is performed
    - find an appropriate random variable X to represent the outcome
- others: smallest enclosing circle, minimum cut, primality test
- perfect random number: using fresh start of computer and taking bits from the picoseconds or quantum physicsx

Freivald's algorithm

- input: given matrices A, B, C
- output: true if $A \cdot B = C$, else false
- fastest deterministic algorithm: using Strassen's method $\Theta(n^{log_2 7})$
- algorithm $O(kn^2)$
    - pick uniformly random bit vector of size n, $\hat{x}$
    - check if $A \cdot (B \cdot \hat{x}) = C \cdot \hat{x}$
    - repeat k times with independent $\hat{x}$
    - if all succeed, then true, else false
- error probability: at most 2^{-k} per pass

finding approximate median

- deterministic: $O(n)$ (quick select)
- approximate median: element y which has rank between $n/4$ and $3n/4$
- randomly pick an element, the probability of being an approximate median is at least $1/2$
    - repeat this k times to increase probability of success

- probability of error: less than $\frac{1}{n^2}$ if $k = 1 + 10\log n$

analysis: balls in bins

> 📌 use an indicator variable of the event and find the sum of that instead
>
> if event involves multiple variables, compute the summation across all variables

- given m balls to be placed in n bins with uniformly random probability
- probability k bins are empty: $(1 - \frac{k}{n})^m$
- probability at least 1 bin is empty: $OBE(n, m) = \binom{n}{1}(1 - \frac{1}{n})^m - \binom{n}{2}(1 - \frac{2}{n})^m + \ldots (-1)^{k+1}\binom{n}{k}(1 - \frac{k}{n})^m + \ldots$
- expected number of empty bins: let X_i be random variable such that
    - $E(X_i) = 1 \times P(i^{th} \text{ bin is empty}) + 0 \times P(i^{th} \text{ bin is not empty}) = (1 - \frac{1}{n})^m$

$$X_i = \begin{cases} 1, & \text{if } i^{th} \text{ bin is empty} \\ 0 \end{cases}$$

dynamic programming

- overlapping subproblem: recursive solution contains a "small" number of distinct subproblems repeated many times
- optimal substructure: optimal solution of a state can be constructed from the optimal solution of subproblems
- cut-and-paste argument (extension of proof by contradiction)
    - suppose an "optimal" solution is found with suboptimal substructures
    - cut the suboptimal substructures out and paste the optimal substructure to reveal an even more optimal solution
    - therefore, there is a contradiction
- top-down vs bottom-up

- both work
- top-down sometimes saves some computation of unnecessary subproblems but can introduce overhead of recursive call
- both provide same asymptotic time complexity
- top-down may suffer from space overhead of recursive call and space optimization is harder

- longest common subsequence, $lcs(i,j)$ is the longest common subsequence of $A[:i]$ and $B[:j]$A

$$lcs(i,j) = \begin{cases} lcs(i-1,j-1)+1, A[i]=B[j] \\ \max\{lcs(i-1,j), lcs(i,j-1)\}, A[i] \neq B[j] \\ 0, i=0 \lor j=0 \end{cases}$$

$$T(n) \in \Theta(nm)$$

- longest palindromic subsequence, $dp(i,j)$ is the longest palindromic subsequence between $A[i:j]$
  - extension of LCS: run LCS on original and reversed string and find the overlap
  - optimal solution:

$$dp(i,j) = \begin{cases} dp(i+1,j-1)+1, A[i]=B[j] \\ \max\{dp(i+1,j), dp(i,j-1)\}, A[i] \neq B[j] \\ 0, i=0 \lor j=0 \end{cases}$$

$$T(n) \in \Theta(nm)$$

knapsack problem

- input: $(w_1,v_1),(w_2,v_2),\ldots,(w_n,v_n)$ and W
- output: subset of $\{1,2,\ldots,n\}$ such that $\sum_{i\in S} v_i : \sum_{i \in S} w_i \leq W$

$$dp(i,j) = \begin{cases} 0, i=0 \lor j=0 \\ \max\{dp(i-1,j), dp(i-1,j-w_i)+v_i\}, w_i \leq j \\ dp(i-1,j) \end{cases}$$

$$T(n) \in \Theta(nW)$$

- $dp(i, j)$ is the maximum value achievable given $items[: i]$ items and j maximum W
- if infinite supply of weights, modify the algorithm so once a weight is taken, we can continue to take from it by going to $dp(i, j - w_i)$

math revision

exponentials

- $a^{-1} = \frac{1}{a}$
- $(a^m)^n = a^{mn}$
- $a^m \times a^n = a^{m+n}$
- $e^x \geq 1 + x$
- exponentials of different bases differ by an exponential factor (cannot be ignored)

📌 any exponential function with base $a > 1$ grows faster than any polynomial

lemma: for any constants $k > 0$ and $a > 1$, $n^k = o(a^n)$

logarithms

- binary log: $\lg n = \log_2 n$
- natural log: $\ln n = \log_e n$
- exponentiation: $\lg^k n = (\lg n)^k$
- composition: $\lg \lg n = \lg(\lg n)$
- $a = b^{\log_b a}$
- $\log_c(ab) = \log_c a + \log_c b$

- $\log_b a^n = n \log_b a$
- $\log_b a = \dfrac{\log_c a}{\log_c b}$
- $\log_b (1/a) = -\log_b a$
- $\log_b a = \dfrac{1}{\log_a b}$
- $a^{\log_b c} = c^{\log_b a}$
- base of logarithm does not matter in asymptotic analysis

Stirling's approximation

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

$$\log(n!) = \Theta(n \lg n)$$

summations

- arithmetic series
  - $a_n = a_1 + (n-1) \times d$
  - $S_n = \frac{n}{2}(2a_1 + (n-1) \times d) = \frac{n \times (a_1 + a_n)}{2} \in \Theta(n^2)$
- geometric series
  - $g_n = g_1 \times r^{n-1}$
  - $S_n = \dfrac{a(1-r^n)}{1-r}$
  - $S_\infty = \dfrac{a}{1-r}$ when $|x| < 1$
- harmonic series
  - $H_n = 1 + 1/2 + 1/3 + \ldots + 1/n = \sum_{k=1}^{n} 1/k = \ln n + O(1)$

common algorithms

fibonacci

recursive

```
def fib(n):
    if n == 0: return 0
    if n == 1: return 1
    return fib(n - 1) + fib(
```

$T(n) = T(n-1) + T(n-2) + O(1)$

$T(n) \in O(2^n)$

more precisely: $O(\phi^n)$

iterative

```
def fib(n):
    if n == 0: return 0
    if n == 1: return 1
    p2, p1 = 0, 1
    for i in range(2, n + 1)
        p2, p1 = p1, p1 + p2
    return p1
```

$T(n) \approx 5n$

using matrix multiplication

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} F_n + F_{n-1} & F_n \\ F_{n-1} + F_{n-2} & F_{n-1} \end{pmatrix} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

merge sort

```
def merge_sort(arr):
    if len(arr) == 1: return arr
    left = merge_sort(arr[:len(arr)//2])
    right = merge_sort(arr[len(arr)//2:])
    return merge(left, right)
```

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n), n > 1 \\ \Theta(1), n = 1 \end{cases}$$

powering a number

- find $F(a, n) = a^n$ (may use $a^n \% m$ to avoid large numbers)

$$F(a,n) = \begin{cases} F(a, \lfloor \frac{n}{2} \rfloor)^2, n\&1 = 0 \\ F(a, \lfloor \frac{n}{2} \rfloor)^2 * F(a,1), n\&1 = 1 \end{cases}$$

- $T(n) = T(n/2) + \Theta(1) \in \Theta(\log n)$
- used to calculate fibonacci numbers in $\Theta(\log n)$ time

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \psi^n)$$

matrix multiplication

naive $\Theta(n^3)$

```
# Outer left dimension
for i in range(n):
    # Outer right dimension
    for j in range(n):
        # Inner dimension
        for k in range(n):
            C[i][j] += A[i][k] * B[k][j]
```

Strassen's method $\Theta(n^{log_2 7})$

- leverages the fact that matrix addition is faster than matrix multiplication
- divide matrix into quadrants of size $n/2 \times n/2$

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

- original equations: $T(n) = 8T(n/2) + \Theta(n^2)$
    - $r = ae + bg$
    - $s = af + bh$
    - $t = ce + dg$
    - $u = cf + dh$

- Strassen's equations: $T(n) = 7T(n/2) + \Theta(n^2)$ with 1 less matrix multiplication
    - $P_1 = a \times (f - h)$
    - $P_2 = (a + b) \times h$
    - $P_3 = (c + d) \times e$
    - $P_4 = d \times (g - e)$
    - $P_5 = (a + d) \times (e + h)$
    - $P_6 = (b - d) \times (g + h)$
    - $P_7 = (a - c) \times (e + f)$
    - $r = P_5 + P_4 - P_2 + P_6$
    - $s = P_1 + P_2$
    - $t = P_3 + P_4$
    - $u = P_5 + P_1 - P_3 - P_7$