CS3230
AY23/24 SEM 2

01. ASYMPTOTIC ANALYSIS

- **algorithm** $\rightarrow$ a *finite* sequence of well-defined instructions to solve a given computational problem
- **word-RAM model** $\rightarrow$ runtime is the total number of instructions executed
  - operators, comparisons, if, return, etc
  - each instruction operates on a *word* of data (limited size) $\Rightarrow$ fixed constant amount of time

Asymptotic Notations

upper bound ($\leq$): $f(n) = O(g(n))$
if $\exists c > 0, n_0 > 0$ such that $\forall n \geq n_0$,
$\boxed{0 \leq f(n) \leq cg(n)}$

lower bound ($\geq$): $f(n) = \Omega(g(n))$
if $\exists c > 0, n_0 > 0$ such that $\forall n \geq n_0$,
$\boxed{0 \leq cg(n) \leq f(n)}$

tight bound: $f(n) = \Theta(g(n))$
if $\exists c_1, c_2, n_0 > 0$ such that $\forall n \geq n_0$,
$\boxed{0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)}$

o**-notation** ($<$): $f(n) = o(g(n))$
if $\forall c > 0, \exists n_0 > 0$ such that $\forall n \geq n_0$,
$\boxed{0 \leq f(n) < cg(n)}$

ω**-notation** ($>$): $f(n) = \omega(g(n))$
if $\forall c > 0, \exists n_0 > 0$ such that $\forall n \geq n_0$,
$\boxed{0 \leq cg(n) < f(n)}$

Limits

Assume $f(n), g(n) > 0$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \qquad \Rightarrow f(n) = o(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \qquad \Rightarrow f(n) = O(g(n))$$
$$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \qquad \Rightarrow f(n) = \Theta(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \qquad \Rightarrow f(n) = \Omega(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \qquad \Rightarrow f(n) = \omega(g(n))$$

Proof. using delta epsilon definition

Properties of Big O

$\boxed{\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))}$

- **transitivity** - applies for $O, \Theta, \Omega, o, \omega$
  $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- **reflexivity** - for O, Ω, Θ, $\quad f(n) = O(f(n))$
- **symmetry** - $f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$
- **complementarity** -
  - $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$
  - $f(n) = o(g(n)) \iff g(n) = \omega(f(n))$
- **misc**
  - if $f(n) = \omega(g(n))$, then $f(n) = \Omega(g(n))$
  - if $f(n) = o(g(n))$, then $f(n) = O(g(n))$

$\boxed{\log\log n < \log n < (\log n)^k < n^k < k^n}$

insertion sort: $O(n^2)$ with worst case $\Theta(n^2)$

02. SOLVING RECURRENCES

for a sub-problems of size $\frac{n}{b}$ where $f(n)$ is the time to divide and combine,
$$T(n) = aT(\tfrac{n}{b}) + f(n)$$

Telescoping method

The telescoping method uses the telescoping series
For any sequence $a_1, a_2, \ldots, a_n$,
$\sum_{k=0}^{n-1} a_k - a_{k+1} =$
$(a_0 - a_1) + (a_1 - a_2) + \ldots + (a_{n-2} - a_{n-1}) + (a_{n-1} - a_n)$
$= a_0 - a_n$

example

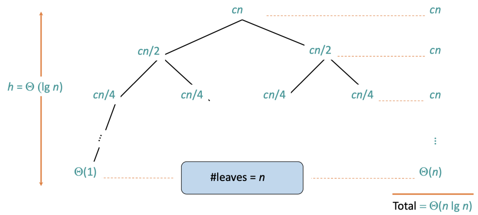
Proof. $T(n) = 2T(n/2) + n \Rightarrow \Theta(n \lg n)$

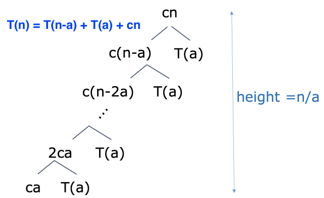
$T(n) = 2T(n/2) + n$
$\Rightarrow \frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$ (Divide by n)
By telescoping method, we have ...
$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$
$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1$
$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1$
...
$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$
Using property of telescoping series, we have
$\frac{T(n)}{n} = \frac{T(1)}{1} + \lg n$ (Height = $\lg n$)
$T(n) = n \cdot T(1) + n \lg n \in \theta(n \lg n)$

Recursion tree

total = height $\times$ number of leaves

- each node represents the cost of a single subproblem
- height of the tree = longest path from root to leaf





Recursion tree is useful for visualising recurrences.

Master method

$a \geq 1, b > 1$, and f is asymptotically positive
$T(n) = aT(\frac{n}{b}) + f(n) =$
$$\begin{cases} \Theta(n^{\log_b a}) & \text{if } f(n) < n^{\log_b a} \text{ polynomially} \\ \Theta(n^{\log_b a} \log n) & \text{if } f(n) = n^{\log_b a} \\ \Theta(f(n)) & \text{if } f(n) > n^{\log_b a} \text{ polynomially} \end{cases}$$

three common cases

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$,
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ by n^ϵ factor.
   - then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$,
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.
   - then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$,
   - and $f(n)$ satisfies the **regularity condition**
     - $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n
     - this guarantees that the sum of subproblems is smaller than $f(n)$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ by n^ϵ factor
   - then $T(n) = \Theta(f(n))$.

Substitution method

1. guess that $T(n) = O(f(n))$.
2. verify by induction:
   - 2.1. to show that for $n \geq n_0, T(n) \leq c \cdot f(n)$
   - 2.2. set $c = \max\{2, q\}$ and $n_0 = 1$
   - 2.3. verify base case(s): $T(n_0) = q$
   - 2.4. recursive case ($n > n_0$):
     - by strong induction, assume $T(k) \leq c \cdot f(k)$ for $n > k \geq n_0$
     - T(n) = ¡recurrence¿ ... $\leq c \cdot f(n)$
   - 2.5. hence $T(n) = O(f(n))$.

! may not be a tight bound!

example

Proof. $T(n) = 4T(n/2) + n^2/\lg n \Rightarrow \Theta(n^2 \lg\lg n)$

$T(n) = 4T(n/2) + \frac{n^2}{\lg n}$
$= 4(4T(n/4) + \frac{(n/2)^2}{\lg n - lg 2}) + \frac{n^2}{\lg n}$
$= 16T(n/4) + \frac{n^2}{\lg n - \lg 2} + \frac{n^2}{\lg n}$
$= \sum_{k=1}^{\lg n} \frac{n^2}{\lg n - k}$
$= n^2 \lg\lg n$ by approx. of harmonic series ($\sum \frac{1}{k}$)

Proof. $T(n) = 4T(n/2) + n \Rightarrow O(n^2)$

To show that for all $n \geq n_0$, $T(n) \leq c_1 n^2 - c_2 n$

1. Set $c_1 = q + 1, c_2 = 1, n_0 = 1$.
2. Base case ($n = 1$): subbing into $c_1 n^2 - c_2 n$,
   $T(1) = q \leq (q+1)(1)^2 - (1)(1)$
3. Recursive case ($n > 1$):

- by strong induction, assume $T(k) \leq c_1 \cdot k^2 - c_2 \cdot k$ for all $n > k \geq 1$
- $T(n) = 4T(n/2) + n$
  $= 4(c_1(n/2)^2 - c_2(n/2)) + n$
  $= c_1 n^2 - 2c_2 n + n$
  $= c_1 n^2 - c_2 n + (1 - c_2)n$
  $= c_1 n^2 - c_2 n$ since $c_2 = 1 \Rightarrow 1 - c_2 = 0$

$\square$

03. ITERATION, RECURSION, DIVIDE-AND-CONQUER

Iterative Algorithms

- **iterative** $\rightarrow$ loop(s), sequentially processing input elements
- **loop invariant** implies correctness if
  - *initialisation* - true before the first iteration of the loop
  - *maintenance* - if true before an iteration, it remains true at the beginning of the next iteration
  - *termination* - true when the algorithm terminates

examples

- **insertionSort**: with loop variable as j, $A[1..J-1]$ is sorted.
- **selectionSort**: with loop variable as j, the array $A[1..j-1]$ is sorted and contains the $j-1$ smallest elements of A.

Divide-and-Conquer

Powering a Number

problem: compute $f(n, m) = a^n \pmod m$ for all $n, m \in \mathbb{Z}$
- observation: $f(x+y, m) = f(x, m) * f(y, m) \pmod m$
- **naive solution**: recursively compute and combine $f(n-1, m) * f(1, m) \pmod m$
  - $T(n) = T(n-1) + T(1) + \Theta(1) \Rightarrow T(n) = \Theta(n)$
- **better solution**: divide and conquer
  - divide: trivial
  - conquer: recursively compute $f(\lfloor n/2 \rfloor, m)$
  - combine:
    - $f(n, m) = f(\lfloor n/2 \rfloor, m)^2 \pmod m$ if n is even
    - $f(n, m) = f(1, m) * f(\lfloor n/2 \rfloor, m)^2 \pmod m$ if odd
  - $T(n) = T(n/2) + \Theta(1) \Rightarrow \Theta(\log n)$

Fibonacci Numbers

- The recursive algorithm $F(n) = F(n-1) + F(n-2)$ to get the n-th Fibonacci number is $O(2^n)$
- The iterative Fibonacci algorithm runs n $O(n)$.
- We can use the powering method to get the n-th Fibonacci algorithm in $\theta(\log n)$.
- $\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} F_n + F_{n-1} & F_n \\ F_{n-1} + F_{n-2} & F_{n-1} \end{pmatrix}$
- $= \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
- Thus, we have
- $\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$

G(n)

1. If n is even, $G(n) = G(\frac{n}{2}) \cdot G(\frac{n}{2})$
2. Otherwise n is odd, $G(n) = G(\frac{n}{2}) \cdot G(\frac{n}{2}) \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
3. Extract F_n from the answer $G(n)$.

Analysis: Matrix Multiplication Fibonacci

1. Dividing and combining takes $O(1)$ time.
2. Recurrence relation: $T(n) = T(\frac{n}{2}) + \theta(1)$
3. Hence time taken is $\theta(\log n)$

Strassen's Matrix Multiplication

- Standard matrix multiplication algorithm takes $\theta(n^3)$ time

```
MAT-MULT(A, B)
  Initialize C[i][j]
  For i = 1 to n
    For j = 1 to n
      C[i][j] = 0
      For k = 1 to n
        C[i][j] = c[i][j] + A[i][k] * B[k][j]
  return C
```

- Strassen's smart idea is shown below
- $\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix}$
- $P_1 = a \cdot (f - h)$
- $P_2 = (a + b) \cdot h$
- $P_3 = (c + d) \cdot e$
- $P_4 = d \cdot (g - e)$
- $P_5 = (a + d) \cdot (e + h)$
- $P_6 = (b - d) \cdot (g + h)$
- $P_7 = (a - c) \cdot (e + f)$
- $r = P_5 + P_4 - P_2 + P_6$
- $s = P_1 + P_2$
- $t = P_3 + P_4$
- $u = P_5 + P_1 - P_3 - P_7$
- Strassen's matrix multiplication needs 7 multiplication of matrices = of size $\frac{n}{2}$ and 18 additions.
- Divide: Divide A and B into $(\frac{n}{2})$ by $(\frac{n}{2})$ submatrices.
- Conquer: Perform 7 matrix multiplications of size $\frac{n}{2}$.
- Combine: Do the additions / subtractions as mentioned.
- Recurrence Relation: $7T(\frac{n}{2}) + \theta(n^2)$
- By master theorem, $T(n) = \theta(n^{\log_2 7}) \approx \theta(n^{2.81})$

04. SORTING ANALYSIS

a. Comparison based sorting (Lower bound)

The best worst case running time for comparison based sorting is $O(n \log n)$.

- Any comparison based sorting algorithm can be modelled using a decision tree.
- One tree for each input of size n. (n element to be sorted)
- View each branch of the tree (from root to a leaf) as the comparisons done by the algorithm based on the results of earlier comparisons.
- Leaves denote the sorted list output by the algorithm based on the results of the comparisons done in the corresponding branch.
- Worst case running time (number of comparisons done) is the longest path from root to leaf.

Proof. Any comparison based sorting algorithm takes at least $\Omega(n \log n)$ time.

1. Model the algorithm as a tree. The tree must contain at least $n!$ leaves for every possible permutation.
2. The height of the binary tree is thus at least $\log(n!)$
3. $\log(n!) = n \log n + O(\log n)$ (Stirling's)
4. $\log(n!) \in \Omega(n \log n)$

b. Average Case Analysis

- **average case** $A(n) \to$ expected running time when the input is chosen uniformly at random from the set of all $n!$ permutations
  - $A(n) = \frac{1}{n!} \sum_{\pi} Q(\pi)$ where $Q(\pi)$ is the time complexity when the input is permutation π.
  - $A(n) = \underset{x \sim \mathcal{D}_n}{\mathbb{E}}$ [Runtime of Alg on x]
    - $\mathbb{E}_{x \sim \mathcal{D}_n}$ is a probability distribution on U restricted to inputs of size n.

Quicksort Analysis

- divide & conquer, linear-time $\Theta(n)$ partitioning subroutine
- assume we select the first array element as pivot
- $T(n) = T(j) + T(n - j - 1) + \Theta(n)$
  - if the pivot produces subarrays of size j and $(n - j - 1)$
- **worst-case**: $T(n) = T(0) + T(n-1) + \Theta(n) \Rightarrow \Theta(n^2)$

Proof. for quicksort, $A(n) = O(n \log n)$

let $P(i)$ be the set of all those permutations of elements $\{e_1, e_2, \ldots, e_n\}$ that begins with e_i.

Let $G(n, i)$ be the average running time of quicksort over $P(i)$. Then

$G(n) = A(i-1) + A(n-i) + (n-1)$.

$A(n) = \frac{1}{n} \sum_{i=1}^{n} G(n, i)$

$= \frac{1}{n} \sum_{i=1}^{n} (A(i-1) + A(n-i) + (n-1))$

$= \frac{2}{n} \sum_{i=1}^{n} A(i-1) + n - 1$

$= O(n \log n)$ by taking it as area under integration

quicksort vs mergesort

	average	best	worst
quicksort	$1.39 n \lg n$	$n \lg n$	$n(n-1)$
mergesort	$n \lg n$	$n \lg n$	$n \lg n$

- disadvantages of mergesort:
  - overhead of temporary storage
  - cache misses
- advantages of quicksort
  - in place
  - reliable (as $n \uparrow$, chances of deviation from avg case $\downarrow$)
- issues with quicksort
  - **distribution-sensitive** $\to$ time taken depends on the initial (input) permutation

c. Linear Time Sorting

Counting Sort

- No comparisons made between elements
- Input array: $A[1..n]$, where $A[i] \in 1, 2, \ldots, k$
- Output array: $B[1..n]$ (sorted)
- Use $C[1..k]$ for intermediate steps
- If $k = O(n)$, then counting sort takes $\theta(n)$ time.

```
for i = 1 to k:
  C[i] = 0
for j = 1 to n:
  C[A[j]] = C[A[j]] + 1
for i = 2 to k:
  C[i] = C[i] + C[i - 1]
for j = n downto 1:
  B[C[A[j]]] = A[j]
  C[A[j]] = C[A[j]] - 1
```

Radix Sort

```
Suppose there are T digits.
for i = 1 to T:
  sort by the ith least significant bit
  using counting sort
```

- T passes
- Each pass takes $\theta(n + k)$ time, where numbers are between 1 to k
- If $b-$bit word is broken into $\frac{b}{r}$ groups of r bit words, then
  - There are $\frac{b}{r}$ passes
  - Each pass takes $\theta(n + 2^r)$
  - Total time: $\theta(\frac{b}{r}(n + 2^r))$
- Choose r to minimize the above. Optimal r is about $\log n$
- $2^{\log n} = n; \theta(\frac{b}{\log n} \cdot 2n) = \theta(\frac{bn}{\log n})$
- If the numbers are in the range 1 to n^d, then $b = d \log n$. Radix sort runs in $\theta(dn)$.

Correctness of Radix Sort

- Prove by induction that when we have sorted the least significant t digits, then the numbers are sorted according to their values on the least significant $t-$digits.
- $P(1)$ holds. The first pass of radix sort will sort based on the least significant digit.
- Suppose $P(k-1)$ holds. Then show $P(k)$:
- Clearly the numbers are sorted based on the k-th least significant digit by the $k-$th pass.
- **Due to stable sort**, within the same $k-$th least significant digit, **the algorithm doesn't change their relative position!**
- Thus, the numbers are sorted within the groups of having the same $k-$th least significant digit.

05.RANDOMISED ALGORITHMS

- **randomised algorithms** $\to$ output and running time are **functions** of the **input** and **random bits chosen**
  - vs non-randomised: output & running time are functions of the *input only*
- expected running time = worst-case running time = $E(n) = \underset{\text{input } x \text{ of size } n}{\max} \mathbb{E}$[Runtime of RandAlg on x]
- **randomised quicksort**: choose pivot at random
  - probability that the runtime of *randomised* quicksort exceeds average by $x\% = n^{-\frac{x}{100} \ln \ln n}$
  - P(time takes at least double of the average) = 10^{-15}
  - distribution insensitive

Randomised Quicksort Analysis

$T(n) = n - 1 + T(q-1) + T(n-q)$

Let $A(n) = \mathbb{E}[T(n)]$ where the expectation is over the randomness in expectation.

Taking expectations and applying linearity of expectation:

$A(n) = n - 1 + \frac{1}{n} \sum_{q=1}^{n} (A(q-1) + A(n-q))$

$= n - 1 + \frac{2}{n} \sum_{q=1}^{n-1} A(q)$

$A(n) = n \log n \quad \Rightarrow$ same as average case quicksort

Randomised Quickselect

- $O(n)$ to find the k^{th} smallest element
- randomisation: unlikely to keep getting a bad split

Types of Randomised Algorithms

- randomised **Las Vegas** algorithms
  - output is always correct
  - runtime is a *random variable*
  - e.g. randomised quicksort, randomised quickselect
- randomised **Monte Carlo** algorithms
  - output may be incorrect with some small probability
  - runtime is *deterministic*

examples

- *smallest enclosing circle:* given n points in a plane, compute the smallest radius circle that encloses all n points
  - best **deterministic** algorithm: $O(n)$, but complex
  - Las Vegas: average $O(n)$, simple solution
- *minimum cut:* given a connected graph G with n vertices and m edges, compute the smallest set of edges whose removal would disconnect G.
  - best **deterministic** algorithm: $O(mn)$
  - **Monte Carlo**: $O(m \log n)$, error probability n^{-c} for any c
- *primality testing:* determine if an n bit integer is prime
  - best **deterministic** algorithm: $O(n^6)$
  - **Monte Carlo**: $O(kn^2)$, error probability 2^{-k} for any k

Geometric Distribution

Let X be the number of trials repeated until success. X is a random variable and follows a geometric distribution with probability p.

Expected number of trials, $E[X] = \frac{1}{p}$

$Pr[X = k] = q^{k-1} p$

Linearity of Expectation

For any two events X, Y and a constant a,

$E[X + Y] = E[X] + E[Y]$

$E[aX] = aE[X]$

Coupon Collector Problem

n types of coupon are put into a box and randomly drawn with replacement. What is the expected number of draws needed to collect at least one of each type of coupon?

- let T_i be the time to collect the i-th coupon after the $i-1$ coupon has been collected.
  - Probability of collecting a new coupon, $p_i = \frac{(n-(i-1))}{n}$
  - T_i has a **geometric distribution**
  - $E[T_i] = 1/p_i$

- total number of draws, $T = \sum_{i=1}^{n} T_i$
- $E[T] = E[\sum_{i=1}^{n} T_i] = \sum_{i=1}^{n} E[T_i]$ by linearity of expectation
  $= \sum_{i=1}^{n} \frac{n}{n-(i-1)} = n \cdot \sum_{i=1}^{n} \frac{1}{i} = \Theta(n \lg n)$

06. DYNAMIC PROGRAMMING

- **cut-and-paste proof** $\rightarrow$ proof by contradiction - suppose you have an optimal solution. Replacing ("cut") subproblem solutions with this subproblem solution ("paste" in) should improve the solution. If the solution doesn't improve, then it's not optimal (contradiction).
- **overlapping subproblems** - recursive solution contains a small number of distinct subproblems repeated many times

Longest Common Subsequence

- for sequence $A : a_1, a_2, \ldots, a_n$ stored in array
- C is a **subsequence** of $A \rightarrow$ if we can obtain C by removing zero or more elements from A.

problem: given two sequences $A[1..n]$ and $B[1..m]$, compute the *longest* sequence C such that C is a subsequence of A and B.

brute force solution

- check *all* possible subsequences of A to see if it is also a subsequence of B, then output the longest one.
- analysis: $O(m2^n)$
  - checking each subsequence takes $O(m)$
  - 2^n possible subsequences

recursive solution

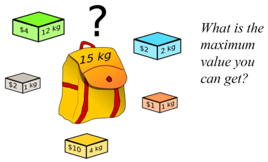
let $LCS(i, j)$: longest common subsequence of $A[1..i]$ and $B[1..j]$

- base case: $LCS(i, 0) = \emptyset$ for all i, $LCS(0, j) = \emptyset$ for all j
- general case:
  - if last characters of A, B are $a_n = b_m$, then $LCS(n, m)$ must terminate with $a_n = b_m$
    - the optimal solution will match a_n with b_m
  - if $a_n \neq b_m$, then either a_n or b_m is not the last symbol
- **optimal substructure**: (general case)
  - if $a_n = b_m$, $LCS(n, m) = LCS(n-1, m-1) :: a_n$
  - if $a_n \neq b_m$, $LCS(n, m) = LCS(n-1, m) \mid\mid LCS(n, m-1)$
- **simplified problem**:
  - $L(n, m) = 0$ if $n = 0$ or $m = 0$
  - if $a_n = b_m$, then $L(n, m) = L(n-1, m-1) + 1$
  - if $a_n \neq b_m$, then $L(n, m) = \max(L(n, m-1), L(n-1, m))$

  analysis
  - number of distinct subproblems $= (n+1) \times (m+1)$
  - to use $O(\min\{m, n\})$ space: bottom-up approach, column by column
  - memoize for DP $\Rightarrow$ makes it $O(mn)$ instead of exponential time

Knapsack Problem

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



- 2^n subsets $\Rightarrow$ naive algorithm is costly
- **recursive solution**:
  - let $m[i, j]$ be the maximum value that can be obtained using a subset of items $\{1, 2, \ldots, i\}$ with total weight no more than j.
  - $m[i, j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ \max\{m[i-1, j-w_i] + v_i, m[i-1, j]\}, & \text{if } w_i \leq j \\ m[i-1, j], & \text{otherwise} \end{cases}$
- **analysis**: $O(nW)$
  - **!** $O(nW)$ is **not** a polynomial time algorithm
  - not polynomial in input bitsize
    - W can be represented in $O(\lg W)$ bits
    - n can be represented in $O(\lg n)$ bits
  - polynomial time is strictly in terms of the number of bits for the input

Changing Coins

problem: use the fewest number of coins to make up n cents using denominations $d_1, d_2, \ldots, d_n$. Let $M[j]$ be the fewest number of coins needed to change j cents.

- **optimal substructure**:
  - $M[j] = \begin{cases} 1 + \min_{i \in [k]} M[j - d_i], & j > 0 \\ 0, & j = 0 \\ \infty, & j < 0 \end{cases}$

  Proof. Suppose $M[j] = t$, meaning $j = d_{i_1} + d_{i_2} + \cdots + d_{i_t}$ for some $i_1, \ldots, i_t \in \{1, \ldots, k\}$.
  Then, if $j' = d_{i_1} + d_{i_2} + \cdots + d_{i_{t-1}}$, $M[j'] = t - 1$, because otherwise if $M[j'] < t - 1$, by **cut-and-paste** argument, $M[j] < t$.

- runtime: $O(nk)$ for n cents, k denominations

07. GREEDY ALGORITHMS

- solve only one subproblem at each step
- beats DP and divide-and-conquer when it works
- **greedy-choice property** $\rightarrow$ a locally optimal choice is globally optimal

Examples

Fractional Knapsack

- $O(n \log n)$
- **greedy-choice property**: let j^* be the item with *maximum* value/kg, v_j / w_i. Then there exists an optimal knapsack containing $\min(w_{j^*}, W)$ kg of item j^*.
- **optimal substructure**: if we remove w kg of item j from the optimal knapsack, then the remaining load must be the optimal knapsack weighing at most $W - w$ kgs that one can take from $n - 1$ original items and $w_j - w$ kg of item j.

Proof. cut-and-paste argument

Suppose the remaining load after removing w kgs of item j was *not* the optimal knapsack weighing ...

Then there is a knapsack of value $> X - v_j \cdot \frac{w}{w_j}$ with weight ...

Combining this knapsack with w kg of item j gives a knapsack of value $> X \Rightarrow$ contradiction!

Minimum Spanning Trees

for a connected, undirected graph $G = (V, E)$, find a spanning tree T that connects all vertices with minimum weight. Weight of spanning tree T, $w(T) = \sum_{(u,v) \in T} w(u, v)$.

- **optimal substructure**: let T be a MST. remove any edge $(u, v) \in T$. then T is partitioned into T_1, T_2 which are MSTs of $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.

  Proof. cut-and-paste: $w(T) = w(u, v) + w(T_1) + w(T_2)$

  if $w(T_1') < w(T_1)$ for G_1, then $T' = \{(u, v)\} \cup T_1' \cup T_2$ would be a lower-weight spanning tree than T for G.

  $\Rightarrow$ contradiction, T is the MST

- **Prim's algorithm** - at each step, add the least-weight edge from the tree to some vertex outside the tree
- **Kruskal's algorithm** - at each step, add the least-weight edge that does *not* cause a cycle to form

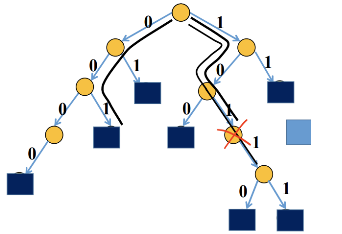
Binary Coding

Given an alphabet set $A : \{a_1, a_2, \ldots, a_n\}$ and a text file F (sequence of alphabets), how many bits are needed to encode a text file with m characters?

- **fixed length encoding**: $m \cdot \lceil \log_2 n \rceil$
  - encode each alphabet to unique binary string of length $\lceil \log_2 n \rceil$
  - total bits needed for m characters $= m \cdot \lceil \log_2 n \rceil$
- **variable length encoding**
  - different characters occur with different frequency $\Rightarrow$ use fewer bits for *more frequent* alphabets
  - average bit length, $ABL(\gamma) = \sum_{x \in A} f(x) \cdot |\gamma(x)|$
  - BUT overlapping prefixes cause indistinguishable characters

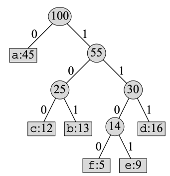
Prefix coding

- a coding $\gamma(A)$ is a **prefix coding** if $\nexists x, y \in A$ such that $\gamma(x)$ is a prefix of $\gamma(y)$.
- **labelled binary tree**: $\gamma(A)$ = label of path from root



- for each prefix code A of n alphabets, there exists a binary tree T on n leaves such that there is a **bijective mapping** between the alphabets and the leaves
- $ABL(\gamma) = \sum_{x \in A} f(x) \cdot |\gamma(x)| = \sum_{x \in A} f(x) \cdot |depth_T(x)|$
- the binary tree corresponding to an *optimal* prefix coding must be a **full binary tree**.
  - every internal node has degree exactly 2
  - multiple possible optimal trees - most optimal depends on alphabet frequencies
- accounting for alphabet **frequencies**:
  - let $a_1, a_2, \ldots, a_n$ be the alphabets of A in non-decreasing order of their frequencies.
  - a_1 must be a leaf node; a_2 *can* be a sibling of a_1.
  - there exists an optimal prefix coding in which a_1 and a_2 are siblings
- derivation of optimal prefix coding: **Huffman's algorithm**
  - keep merging the two least frequent items

```
Huffman(C):
  Q = new PriorityQueue(C)
  while Q:
    allocate a new node z
    z.left = x = extractMin(Q)
    z.right = y = extractMin(Q)
    z.val = x.val + y.val
    Q.add(z)
  return extractMin(Q) // root
```



08. AMORTIZED ANALYSIS

- **amortized analysis** $\rightarrow$ guarantees the *average* performance of each operation in the *worst case*.
- total amortized cost provides an *upper bound* on the total true cost
- For a sequence of n operations $o_1, o_2, \ldots, o_n$,
  - let $t(i)$ be the time complexity of the i-th operation o_i
  - let $f(n)$ be the *worst-case* time complexity for *any* of the n operations
  - let $T(n)$ be the time complexity of all n operations

    $T(n) = \sum_{i=1}^{n} t(i) = nf(n)$

Types of Amortized Analysis

Aggregate method

- look at the whole sequence, sum up the cost of operations and take the average - simpler but less precise
- e.g. binary counter - amortized $O(1)$
- e.g. queues (with `INSERT` and `EMPTY`) - amortized $O(1)$

Accounting method

- charge the i-th operation a fictitious amortized cost $c(i)$

- **amortized cost** $c(i)$ is a fixed cost for each operation
  - **true cost** $t(i)$ depends on when the operation is called
- amortized cost $c(i)$ must satisfy:

$$\sum_{i=1}^{n} t(i) \leq \sum_{i=1}^{n} c(i) \text{ for all } n$$

- take the extra amount for cheap operations early on as "credit" paid in advance for expensive operations
  - **invariant**: bank balance never drops below 0
- the total amortized cost provides an **upper bound** on the total true cost

Potential method

- ϕ : potential function associated with the algo/DS
- $\phi(i)$: potential at the end of the i-th operation
- c_i : amortized cost of the i-th operation
- t_i : true cost of the i-th operation

$$c_i = t_i + \phi(i) - \phi(i-1)$$
$$\sum_{i=1}^{n} c_i = \phi(n) - \phi(0) + \sum_{i=1}^{n} t_i$$

- hence as long as $\phi(n) \geq 0$, then amortized cost is an upper bound of the true cost.

$$\sum_{i=1}^{n} c_i \geq \sum_{i=1}^{n} t_i$$

- usually take $\phi(0) = 0$
- **e.g.** for queue:
  - let $\phi(i)$ = # of elements in queue after the i-th operation
  - amortized cost for insert:
    $c_i = t_i + \phi(i) - \phi(i-1) = 1 + 1 = 2$
  - amortized cost for empty (for k elements):
    $c_i = t_i + \phi(i) - \phi(i-1) = k + 0 - k = 0$
- try to keep $c(i)$ small: using $c(i) = t(i) + \Delta\phi_i$
  - if $t(i)$ is small, we want $\Delta\phi_i$ to be positive and small
  - if $t(i)$ is large, we want $\Delta\phi_i$ to be negative and large

e.g. Dynamic Table (insertion only)

Aggregate method

cost of n insertions =
$$\sum_{i=1}^{n} t(i) \leq n + \sum_{j=1}^{\lfloor \log(n-1) \rfloor} 2^j \leq 3n$$

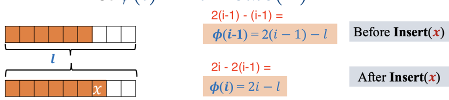
i	1	2	3	4	5	6	7	8	9	10	
$size_i$	1	2	4	4	8	8	8	8	16	16	
$t(i)$	1	1	1	1	1	1	1	1	1	1	Cost for i th insert
		1	2		4				8		Cost for copying due to overflow

Accounting method

- charge $3 per insertion
  - $1 for insertion itself
  - $1 for moving itself when the table expands
  - $1 for moving one of the existing items when the table expands

Potential method

let $\phi(i) = 2i - size(T)$



Operation Insert(x)	Actual Cost	$\Delta\phi_i$	Amortized Cost
Case 1: when table is not full	1	2	3
Case 2: when table is already full	i	$3 - i$	3

Amortized cost of n insertions $= 3n = O(n)$

Actual cost of n insertions $= O(n)$

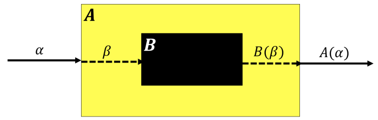
- show that SUM of amortized cost $\geq$ SUM of actual cost
- conclude that sum of amortized cost is $O(f(n)) \Rightarrow$ sum of actual cost is $O(f(n))$

09. REDUCTIONS & INTRACTABILITY

Reduction

Consider two problems A and B, A can be solved as follows:

1. convert instance α of A to an instance of β in B
2. solve β to obtain a solution
3. based on the solution of β, obtain the solution of α.
4. $\Rightarrow$ then we say A **reduces** to B.



instance $\rightarrow$ another word for input

e.g. Matrix Multiplication & Squaring

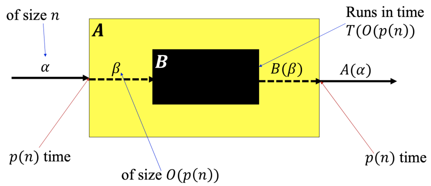
- MAT-MULTI: matrix multiplication
  - *input*: two $N \times N$ matrices A and B.
  - *output*: $A \times B$
- MAT-SQR: matrix squaring
  - *input*: one $N \times N$ matrix C. *output*: $C \times C$
- MAT-SQR can be reduced to MAT-MULTI
  - *Proof*. Given input matrix C for MAT-SQR, let $A = C$ and $B = C$ be inputs for MAT-MULTI. Then $AB = C^2$.
- MAT-MULTI can also be reduced to MAT-SQR!
  - *Proof*. let $C = \begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix}$
    $\Rightarrow C^2 = \begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix} \begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & BA \end{bmatrix}$

T-Sum

- 0-SUM: given array A, output $i, j \in (1, n)$ such that $A[i] + A[j] = 0$
- T-SUM: given array B, output $i, j \in (1, n)$ such that $B[i] + B[j] = T$
- **reduce T-SUM to 0-SUM**:
  - given array B, define array A s.t. $A[i] = B[i] - T/2$.
  - if i, j satisfy $A[i] + A[j] = 0$, then $B[i] + B[j] = T$.

$p(n)$-time Reduction

- **$p(n)$-time Reduction** $\rightarrow$ if for any instance α of problem A of size n,
  - an instance β for B can be constructed in $p(n)$ time
  - a solution to problem A for input α can be recovered from a solution to problem B for input β in time $p(n)$.
- **!** n is in **bits**!
- if there is a $p(n)$-time reduction from problem A to B and a $T(n)$-time algorithm to solve problem B, then there is a $T(O(p(n))) + O(p(n))$ time algorithm to solve A.



- $A \leq_P B$ $\rightarrow$ if there is a $p(n)$-time reduction from A to B for some polynomial function $p(n) = O(n^c)$ for some constant c. ("A is a special case of B")
  - if B has a polynomial time algorithm, then so does A
  - "polynomial time" $\approx$ reasonably efficient
- $A \leq_P B, B \leq_P C \Rightarrow A \leq_P C$

Polynomial Time

- **polynomial time** $\rightarrow$ runtime is polynomial in the **length of the encoding** of the problem instance
- **"standard" encodings**
  - binary encoding of integers
  - list of parameters enclosed in braces (graphs/matrices)
- **pseudo-polynomial** algorithm $\rightarrow$ runs in time polynomial in the **numeric value** if the input but is **exponential** in the **length** of the input
  - e.g. DP algo for KNAPSACK since W is in numeric value
- KNAPSACK is NOT polynomial time: $O(nW \log M)$ but W is not the number of bits
- FRACTIONAL KNAPSACK is polynomial time: $O(n \log n \log W \log M)$

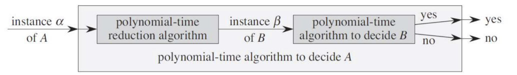
Decision Problems

- **decision problem** $\rightarrow$ a function that maps an instance space I to the solution set $\{YES, NO\}$
- decision vs optimisation problem:
  - **decision problem**: given a directed graph G, *is there* a path from vertex u to v of length $\leq k$?
  - **optimisation problem**: given ..., what is the *length* of the shortest path ... ?
  - convert from **decision** $\rightarrow$ **optimisation**: given an instance of the optimisation problem and a number k, is there a solution with value $\leq k$?
- the decision problem is *no harder than* the optimisation problem.
  - given the optimal solution, check that it is $\leq k$.
  - if we cannot solve the decision problem quickly $\Rightarrow$ then we cannot solve the optimisation problem quickly
- decision $\leq_P$ optimisation

Reductions between Decision Problems

given two decision problems A and B, a polynomial-time reduction from A to B denoted $A \leq_P B$ is a **transformation** from instances α of A and β of B such that

1. α is a *YES*-instance of A $\iff$ β is a *YES*-instance of B
2. the transformation takes polynomial time in the size of α



Examples

- INDEPENDENT-SET: given a graph $G = (V, E)$ and an integer k, is there a subset of $\leq k$ vertices such that no 2 are adjacent?
- VERTEX-COVER: given a graph $G = (V, E)$ and an integer k, is there a subset of $\leq k$ vertices such that each edge is incident to *at least one* vertex in this subset?
- INDEPENDENT-SET $\leq_P$ VERTEX-COVER
  - *Reduction*: to check whether G has an independent set of size k, we check whether G has vertex cover of size $n - k$.

Proof. If INDEPENDENT-SET, then VERTEX-COVER.

Suppose (G, k) is a *YES*-instance of INDEP-SET. Then there is subset S of size $\geq k$ that is an independent set.

$V - S$ is a vertex cover of size $\leq n - k$. Proof: Let $(u, v) \in E$. Then $u \notin S$ or $v \notin S$.

So either u or v is in $V - S$, the vertex cover.

Proof. If VERTEX-COVER, then INDEPENDENT-SET.

Same as above, but flip IS and VC

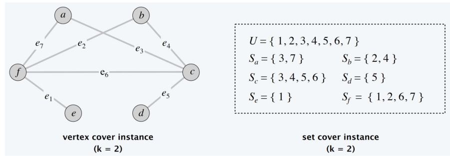
e.g. SET-COVER

Given integers k and n, and collection $\mathcal{S}$ of subsets of $\{1, \ldots, n\}$, are there $\leq k$ of these subsets whose union equals $\{1, \ldots, n\}$?

Claim: **VERTEX-COVER $\leq_P$ SET-COVER**

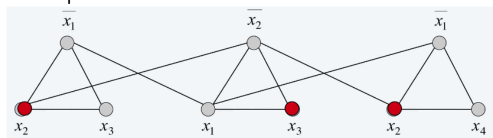
Reduction: given (G, k) instance of VERTEX-COVER, generate an instance $(n, k', \mathcal{S})$ of SET-COVER.

Proof. For each node v in G, construct a set S_v containing all its outgoing edges. (Number each edge)



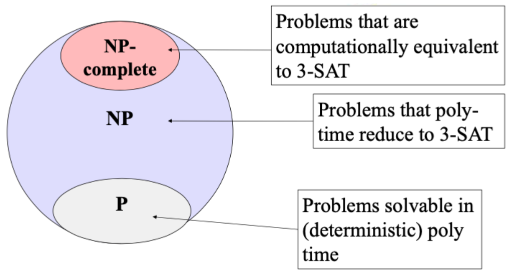
e.g. 3-SAT

- **SAT**: given a CNF formula Φ, does it have a satisfying truth assignment?
  - literal: a boolean variable or its negation $x, \bar{x}$
  - clause: a disjunction (OR) of literals
  - conjunctive normal form (CNF): formula Φ that is a conjunction (AND) of clauses
- **3-SAT** $\rightarrow$ SAT where each clause contains exactly 3 literals
- **3-SAT $\leq_P$ INDEPENDENT-SET**
  - *Reduction*: Construct an instance (G, k) of INDEP-SET s.t. G has an independent set of size k $\iff$ Φ is satisfiable
    - node: each literal term
    - edge: connect 3 literals in a clause in a triangle
    - edge: connect literal to all its negations
    - reduction runs in polynomial time
  - $\Rightarrow$ for k clauses, connecting k vertices form an independent set in G.



10. NP-COMPLETENESS

- **P** $\to$ the class of *decision* problems solvable in (deterministic) polynomial time
- **NP** $\to$ the class of *decision* problems for which polynomial-time verifiable **certificates** of YES-instances exist.
  - aka *non-deterministic polynomial*
  - i.e. no poly-time algo, but verification can be poly-time
  - **certificate** $\to$ result that can be checked in poly-time to verify correctness
- $P \subseteq NP$: any problem in **P** is in **NP**.
  - if $P = NP$, then all these algos can be solved in poly time



NP-Hard and NP-Complete

- a problem A is said to be **NP-Hard** if for *every* problem $B \in NP$, $B \leq_P A$.
  - aka A is at least as hard as every problem in **NP**.
- a problem A is said to be **NP-Complete** if it is in **NP** and is also **NP-Hard**
  - aka the hardest problems in NP.
- **Cook-Levin Theorem** $\to$ every problem in NP-Hard can be poly-time *reduced* to 3-SAT. Hence, **3-SAT is NP-Hard and NP-Complete**.
- NP-Complete problems can still be approximated in poly-time! (e.g. greedy algorithm gives a 2-approximation for VERTEX-COVER)

showing NP-Completeness

1. show that X is in NP. $\Rightarrow$ a YES-instance has a certificate that can be verified in polynomial time
2. show that X is NP-hard
   - by giving a poly-time reduction from another NP-hard problem A to X. $\Rightarrow$ X is at least as hard as A
   - reduction should *not* depend on whether the instance of A is a YES- or NO-instance
3. show that the reduction is valid
   3.1. reduction runs in poly time
   3.2. if the instance of A is a YES-instance, then the instance of X is also a YES-instance
   3.3. if the instance of A is a NO-instance, then the instance of X is also a NO-instance

```
def INDEPENDENT-SET(G, k) -> bool:
    G', k' = reduction(G, k)
    yes_or_no: bool = CLIQUE(G', k') # magically given
    return yes_or_no
```

What to show for a correct reduction:

- `(G, k)` is YES-instance → `(G', k')` is also a YES-instance
- `(G', k')` is YES-instance → `(G, k)` is also a YES-instance
- The transformation takes polynomial time in the size of `(G, k)`

showing NP-HARD

1. take any **NP-Complete** problem A
2. show that $A \leq_P X$

11. ORDER STATISTICS

Selection / Order Statistics

- Given an unsorted list we want to find the i-th smallest element in the list.
- $i = 1$ is minimum and $i = n$ is maximum.
- To get median, we need $i = \lfloor \frac{n+1}{2} \rfloor$ or $i = \lceil \frac{n+1}{2} \rceil$
- **Naive Solution**: Sort and return the i-th element in the sorted list. This takes $\theta(n \log n)$ time.
- Can we do in worst case $O(n)$ time?

What we know

- Selecting the i-th elements require $\geq n$ steps, otherwise there must be some element x which was not seen by our algorithm.
- If we have seen less than n elements, then whatever the order among the other elements, it is not possible to say if x is the i-th smallest element because we can be x above or below other elements.
- We can find the maximum or minimum element in an array of n elements in $\theta(n)$ time and that **naive** algorithm is the best possible.

Linear Time Selection

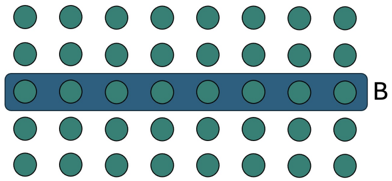
- (Blum, Floyd, Pratt, Rivest, Tarjan, 1973) created the worst case linear time algorithm to select the rank-i element.
- In this problem, we assume that all elements are distinct (otherwise for equal elements distinguish them based on the location they were originally stored.)
- That is consider the element originally at $A[i]$, as $(A[i], i)$.
- Suppose $A[i] = a and A[j] = b$. Then we compare $(a, i) < (b, j)$ if $a < b$ or if $a = b$ then $i < j$.

Select(i, n, A):

1. Divide the array A into $\lceil \frac{n}{5} \rceil$ groups of 5 elements each.
2. Let B be the set of $\lceil \frac{n}{5} \rceil$ elements of the medians of each of the above groups.
3. Recursively find the median x of B by calling $Select(\frac{n}{10}, \frac{n}{5}, B)$
4. Partition A and pivot around x. Let k be the rank of x and A' and A'' be the list of elements $< x$ and $> x$ respectively.
5. If $i = k$, then return x.
6. Else if $i < k$, then return $Select(i, k-1, A')$
7. Else $i > k$, then return $Select(i-k, n-k, A'')$

End

Pivot

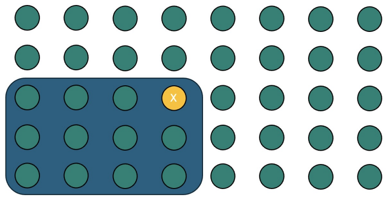


- B = The group of median elements from the $\lceil \frac{n}{5} \rceil$
- There are $\lceil \frac{n}{5} \rceil$ groups of 5 elements.
- For each group of 5 elements, find the median element.
- In total, the size of B is $\lceil \frac{n}{5} \rceil$.

Analysis of Linear Time Selection

1. Steps 1, 2, 4, 5 takes $c_1 n$ time for some constant c_1.
2. Step 3 takes $T(\frac{n}{5})$ time to recursively find the median element of $\frac{n}{5}$ elements.
3. Steps 6 or 7: Note that at least $\lfloor \frac{n}{5} \rfloor$ of the groups have at least 5 elements. (if n is not a multiple of 5)
4. At least $\lfloor \frac{\lfloor \frac{n}{5} \rfloor}{2} \rfloor$ of these group medians are $\leq x$ and respectively $\geq x$.
5. Thus, there are at least $3 \times \lfloor \frac{n}{10} \rfloor$ elements which are at at most / at least x.
6. Thus, Step 6 (or 7) takes at most $T(\frac{7n}{10})$.

Pivot and elements $\leq$ pivot



1. Suppose x was the median element of group of $\lfloor \frac{n}{5} \rfloor$ median elements.
2. Within the group of $\lfloor \frac{n}{5} \rfloor$ median elements, there are $\lfloor \frac{\lfloor \frac{n}{5} \rfloor}{2} \rfloor$ median elements smaller than x.
3. Within the group of median elements, for each median element m, there are 2 elements in m's group that is smaller than m.
4. Hence, in the worst case, there are at most $3 \times \lfloor \frac{n}{10} \rfloor$ elements smaller than x.

Analysis Continued

1. We can write the recurrence relation for the algorithm as follows: $T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + c_1 n$
2. Using the **substitution method**, we take $T(n) < c_2 n$, for some large enough c_2. Take $c_2 > 10c_1$.
3. Base Cases: $n \leq 1000$. This holds for large enough c_2
4. Induction: $c_2(\frac{n}{5}) + c_2(\frac{7n}{10}) + c_1 n \leq c_2 n$
   4.1. $\frac{9c_2 n}{10} + c_1 n \leq c2_n$
   4.2. $c_1 n \leq \frac{c_2 n}{10}$
   4.3. Hence the induction holds by choice of $c_2 > 10c_1$
5. Thus, the i-th element can be found in $\theta(n)$ time.

Helpful Approximations

arithmetic series: $\sum_{k=1}^{n} k = 1+2+3+\cdots+n = \frac{1}{2}\cdot n(n+1)$

geometric series: $\sum_{k=1}^{n} x^k = 1+x+x^2+\cdots+x^n = \frac{x^{n+1}-1}{x-1}$

$\sum_{k=1}^{\inf} x^k = 1+x+x^2+\cdots+x^n = \frac{1}{1-x}$ when $|x|<1$

stirling's approximation: $T(n) = \sum_{i=0}^{n} \log(n-i) = \log\prod_{i=0}^{n}(n-i) = \Theta(n\log n)$

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

$\log(n!) = \theta(n\log n)$

harmonic number, $H_n = \sum_{k=1}^{n}\frac{1}{k} = \Theta(\lg n)$

basel problem: $\sum_{n=1}^{N}\frac{1}{n^2} \le 2-\frac{1}{N} \xrightarrow{N\to\infty} 2$

because $\sum_{n=1}^{N}\frac{1}{N^2} \le 1+\sum_{x=2}^{\log_3 n}\frac{1}{(x-1)x} = 1+\sum_{n=2}^{N}(\frac{1}{n-1}-\frac{1}{n}) = 1+1-\frac{1}{N} = 2-\frac{1}{N}$

number of primes in range $\{1,\ldots,K\}$ is $> \frac{K}{\ln K}$

Logarithm Identities

$a = b^{\log_b a}$

$\log_c ab = \log_c a + \log_c b$

$\log_b a^n = n\log_b a$

$\log_b a = \frac{\log_c a}{\log_c b}$

$\log_b \frac{1}{a} = -\log_b a$

$\log_b a = \frac{1}{\log_a b}$

$a^{\log bc} = c^{\log_b a}$

Base of logarithm does not matter in asymptotics: $\log n = \theta(\ln n) = \theta(\log_{10} n)$

Whereas exponentials of different bases differ by an exponential factor: $4^n = 2^n\cdot 2^n$

Asymptotic Bounds

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < 2^n < 2^{2n}$

$\log_a n < n^a < a^n < n! < n^n$

for any $a, b > 0$, $\quad \log_a n < n^b$

multiple parameters

for two functions $f(m,n)$ and $g(m,n)$, we say that $f(m,n) = O(g(m,n))$ if there exists constants c, m_0, n_0 such that $0 \le f(m,n) \le c\cdot g(m,n)$ for all $m \ge m_0$ or $n \ge n_0$.

set notation

$O(g(n))$ is actually a *set of functions*. $f(n) = O(g(n))$ means $f(n) \in O(g(n))$

- $O(g(n)) = \{f(n) : \exists c, n_0 > 0 \mid \forall n \ge n_0,\ 0 \le f(n) \le cg(n)\}$
- $\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \mid \forall n \ge n_0,\ 0 \le cg(n) \le f(n)\}$
- $\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 \mid \forall n \ge n_0, \quad 0 \le c_1\cdot g(n) \le f(n) \le c_2\cdot g(n)\} = O(g(n)) \cap \Omega(g(n))$
- $o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \mid \forall n \ge n_0, \quad 0 \le f(n) < cg(n)\}$
- $\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \mid \forall n \ge n_0, \quad 0 \le cg(n) < f(n)\}$

example proofs

Proof. that $2n^2 = O(n^3)$

let $f(n) = 2n^2$. then $f(n) = 2n^2 \le n^3$ when $n \ge 2$.

set $c = 1$ and $n_0 = 2$.

we have $f(n) = 2n^2 \le c\cdot n^3$ for $n \ge n_0$. □

Proof. $n = o(n^2)$

For any $c > 0$, use $n_0 = 2/c$.

Proof. $n^2 - n = \omega(n)$

For any $c > 0$, use $n_0 = 2(c+1)$.

Example. let $f(n) = n$ and $g(n) = n^{1+\sin(n)}$.

Because of the oscillating behaviour of the sine function, there is no n_0 for which f dominates g or vice versa.

Hence, we cannot compare f and g using asymptotic notation.

Example. let $f(n) = n$ and $g(n) = n(2+\sin(n))$.

Since $\frac{1}{3}g(n) \le f(n) \le g(n)$ for all $n \ge 0$, then $f(n) = \Theta(g(n))$. (note that limit rules will not work here)

Mentioned Algorithms

- ch.3 - **Euclidean** - efficient computation of GCD of two integers
- ch.3 - **Tower of Hanoi** - $T(n) = 2^n - 1$
  1. move the top $n-1$ discs from the first to the second peg using the third as temporary storage.
  2. move the biggest disc directly to the empty third peg.
  3. move the $n-1$ discs from the second peg to the third using the first peg for temporary storage.
- ch.3 - **MergeSort** - $T(n) = T(\lfloor n/2\rfloor) + T(\lceil n/2\rceil) + \Theta(n)$
- ch.3 - **Karatsuba Multiplication** - multiply two n-digit numbers x and y in $O(n^{\log_2 3})$
  - worst-case runtime: $T(n) = 3T(\lceil n/2\rceil) + \Theta(n)$

Uncommon Notations

- $\perp$ - false

Probability

sample space: S (Example for a dice: $S = 1,2,3,4,5,6$)

event: a subset of the sample space S (Example: Let A be the event we roll an even number from a fair die: $A = 2,4,6$)

Let P denote the **probability distribution** of an event.

- $P(A) \ge 0$
- $P(S) = 1$
- $P(A\cup B) = P(A) + P(B) - P(A\cap B)$ when events A and B are not mutually exclusive.
- $P(A\cup B) = P(A) + P(B)$ for any two mutually exclusive $(P(A\cap B) = \emptyset)$ events A and B.
- $P(A\cap B) = P(A)\cdot P(B)$ when events A and B are independent.
- $P(A|B) = \frac{P(A\cap B)}{P(B)}$ whenever $P(B) \ne 0$
- Bayes theorem: $P(A|B) = \frac{P(A)\cdot P(B|A)}{P(B)} = \frac{P(A)\cdot P(B|A)}{P(A)\cdot P(B|A) + P(A')\cdot P(B|A')}$
- A **random variable** X is a function that maps the sample space S to real numbers.
- The function $f(x) = P(X = x)$ is the probability density function of X.
- Example: When rolling a pair of dice, we let X be the max of the twp values shown on the dice.
  The sample space contains $6^2 = 36$ events.
  $P(X = 3) = 5/36$ because only the elementary events $(1,3), (2,3), (3,3), (3,2), (3,1)$ has the max value 3.
- **Expectation** or **mean** of random variable X is $E(X) = \sum_{x\in S}(x\cdot P(X = x))$
- **Linearity of Expectation**: For any two events X, Y (does not matter whether dependent or independent) and a constant a. $E(X+Y) = E(X) + E(Y)$ $E(aX) = a\cdot E(X)$