CS3230 Condensed Version

by Lee Zong Xun

Useful Mathematical Properties

- Overview

$$1 < \log n < \sqrt{n} < n < n \log n < n^2$$
$$< n^3 < 2^n < 4^n < n! < n^n$$

- $e^x \geq 1 + x$
- Stirling's approximation $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + \theta(\frac{1}{n}))$
- Arithmetic progression

$$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1) = \theta(n^2)$$

- Geometric progression

$$\sum_{k=0}^{n} x^k = \frac{a(x^{n+1}-1)}{x-1}, \text{ if } |x| > 1$$

- Harmonic Series

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$
$$= \ln n + O(1)$$

- Sum of inverse powers of two

$$S_n = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ...$$
$$< 2$$

- Sum of inverse square

$$\sum_{i=1}^{n} \frac{1}{k^2} \leq 1 + \sum_{i=2}^{n} \frac{1}{k(k-1)}$$
$$= 1 + \sum_{i=1}^{n-1} \frac{1}{k(k+1)}$$
$$= 1 + \left(1 - \frac{1}{n-1}\right) < 2$$

- Combinatorics

$$\binom{n+1}{r} = \binom{n}{r} + \binom{n}{r-1}$$
$$\sum_{k=0}^{n} \binom{n}{k} = 2^n \quad (\frac{n}{k})^k \leq \binom{n}{k} \leq (\frac{ne}{k})^k$$

- Boole's Inequality.

$$P[|\bigcup_{i=1}^{\infty} A_i|] \leq \sum_{i=1}^{\infty} P[A_i]$$

- Double factorial

$$n!! \sim \begin{cases} \sqrt{\pi n}\left(\frac{n}{e}\right)^{\frac{n}{2}} & \text{if even} \\ \sqrt{2n}\left(\frac{n}{e}\right)^{\frac{n}{2}} & \text{if odd} \end{cases}$$

- Bayes Theorem

$$Pr[A|B] = \frac{Pr[B|A] \cdot Pr[A]}{Pr[A] \cdot Pr[B|A] + Pr[\bar{A}] \cdot Pr[B|\bar{A}]}$$

- Geometric distribution

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

- Binomial distribution

$$Pr[X = k] = \binom{n}{k} p^k q^{n-k}; E[X] = np$$

- Misc

$$\sum_{i=1}^{n-2} \lg\lg(n-i) = \theta(n \lg\lg n)$$
$$\sum_{i=1}^{\lg n - 1} \lg\lg \frac{n}{a^i} = \theta(\lg n \lg\lg n)$$

Asymptotic Analysis

Comparison Properties

1. Transitivity and Reflexivity
2. Symmetric

$$f(n) = \theta(g(n)) \iff g(n) = \theta(f(n))$$

3. Complementarity

$$f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$$

Useful Facts

1. For any constants $k, d > 0$, $(\log n)^k = o(n^d)$.
2. For any constants $d > 0, u > 1$, $n^d = o(u^n)$.
3. Degree-k polynomials are $O(n^k)$, $o(n^{k+1})$ and $\omega(n^{k-1})$.

Limits Analysis

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \implies f(n) = \Omega(g(n))$
2. $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \implies f(n) = O(g(n))$
3. $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \implies f(n) = \theta(g(n))$
4. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f(n) = o(g(n))$
5. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \implies f(n) = \omega(g(n))$

Notes

1. $f(n) \in o(g(n)) \implies f(n) \in O(g(n))$
2. Let $f(n) = n$ and $g(n) = n^{1+\sin n}$. Then, because of the oscillating behaviour of sine function, there is no n_0 after which f dominates g or f is dominated by g, So, we cannot compare f and g using asymptotic notation.
3. It is possible for same functions to have different asymptotic time complexity depending on the inputs. For example,

$$T(n) = T\left(\frac{n}{2}\right) + n(1 - \cos n)$$

Recurrence

For any recurrence of the form $T(n) \leq T(an) + T(bn) + cn$, if $a + b < 1$, the recurrence will solve to $O(n)$, and if $a + b > 1$, the recurrence is usually equal to $\Omega(nlogn)$.

Telescoping

Given a recurrence relation $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, we want to express as

$$\frac{T(n)}{g(n)} = \frac{T\left(\frac{n}{b}\right)}{g\left(\frac{n}{b}\right)} + h(n), \text{ where } h(n) = \frac{f(n)}{g(n)}$$

Master Theorem

Applies to recurrences of the form $T(n) = aT\frac{n}{b} + f(n)$, where $a \geq 1, b > 1$ and f is asymptotically positive.

Case 1: $f(n) = O\left(n^{\log_b a - \epsilon}\right) \implies T(n) = \theta(n^{\log_b a})$

Case 2: For $d = \log_b a$

- $k \geq 0$, $f(n) = \theta(n^d lg^k n) \implies T(n) = \theta(n^d lg^{k+1} n)$
- $k = -1$, $f(n) = \theta(n^d \lg^k n) \implies T(n) = \theta(n^d \lg\lg n)$
- $k < -1$, $f(n) = \theta(n^d \lg^k n) \implies T(n) = \theta(n^d)$

Case 3: $f(n) = \Omega\left(n^d n^\epsilon\right) \implies T(n) = \theta(f(n))$

Notes

1. if $f(n) \in O(n^{log_b a - \epsilon}) \implies f(n) \in o(n^{log_b a})$
2. Check regularity condition $af(\frac{n}{b}) \leq cf(n)$ where $c < 1$.

Correctness

Correctness of Iterative Algorithms

- **Initialization**: The invariant is true **before** the first iteration of the loop.
- **Maintenance**: If the invariant is true before an iteration, it remains true before the next iteration.
- **Termination**: On terminates, the invariant provides a useful property for showing correctness.

Dijkstra's Algorithm

Termination Criteria: $|R| = |N|$. Each iteration adds one node to the visited set.

The shortest distance from the set X to u:

$$\triangle(X, u) = min\{\delta(s \to v) + w(v, u) \text{ for v in X }\}$$

Invariant:

- All nodes in the visited set has their shortest distance updated correctly.
- For every neighbour v of nodes in the visited set, $d(v) = \triangle(R, v)$

Initialization: Trivially true since the empty set is empty.

Maintenance: Let R' be the set R at the end of the iteration. Then, $R' = R \cup \{u\}$ where u is the minimum distance of all nodes not in R. WTS:

1. $d(u) = \delta(s, u)$ Assume for contradiction, that the distance is not minimum, then there must exists an alternative shorter path. Since $s \in R$, the path Q moves from R to $V \backslash R$. Let (x, y) be the first edge of Q leaving R. The weight of Q is:

$$\delta(s, x) + wt(x, y) + \delta(y, u) < d(u)$$

As y is neighbour of R, by the invariant (2),

$$d(y) = \triangle(R, y) \leq \delta(s, x) + wt(x, y)$$

As edge weights are nonnegative, $d(y) < d(u)$. The algorithm selects u instead of y implies that $d(u) \leq d(y)$, a contradiction.

2. For every neighbour v of u, $d(v) = \triangle(R', v)$ Invariant (2) implies that $d(v) = \triangle(R, v)$ and we just proved that $d(u) = \delta(s, u)$. This shows that it obeys R' too. So $d(v)$ is set correctly, and (2) holds.

Termination:

- After last step, visited set should contain all nodes in the graph.
- By our invariant, the shortest distance from src to any nodes in the graph is determined.

Correctness of Recursive Algorithms

- Use mathematical induction on size of problem
- Prove base cases
- Show that the algorithm works correctly assuming algorithm works correctly for all smaller instances.

Sorting

Comparison-based sort

Any decision tree that can sort n elements must have height $\Omega(nlgn)$

Proof:

The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height-h binary tree has $\leq 2^h$ leaves $\implies n! \leq 2^h$

$$h \geq lg(n!) \quad \text{log is monotonically increasing}$$
$$\geq lg(\frac{n}{e})^n \quad \text{use stirling approximation}$$
$$= nlgn - nlge$$
$$= \Omega(nlgn)$$

Proving the lower requires two steps:

1. Showing that are n possible outcomes
2. Contructing a ternary/binary decision tree with n leaves, and using it to prove the lower bound.

Linear sort

- Counting sort: $O(n + k)$
- Radix sort: $O(\frac{b}{r}(n + 2^r))$

  Assumptions: integers with large range but of few digits such that $k >> n$.

  Steps

  - Sort n words with b bits each. Each word can be viewed as having b/r base- 2^r digits.
  - If each b-bit word is broken into r-bit pieces, each pass of counting sort takes $\theta(n + 2^r)$ time. There are n numbers with range equal to 2^r and there are b/r passes.
  - Increasing r means fewer passes, but as $r > \lg n$, the time grows exponentially in r
  - By differentiation, we determined that the optimal value of r is somewhat slightly smaller than $\lg n$.
  - Choosing $r = \lg n$ implies $T(n, b) = \theta(\frac{bn}{\lg n})$.
  - Suppose numbers range from 0 to $n^d - 1$, we have $b = d \lg n \implies$ radix sort runs in $\theta(dn)$.

  Proof

  - Base case: The radix sort algorithm sorts the 1st digit by stable sorting algorithm. Hence $P(1)$ is true.
  - Inductive case: Assume $P(t-1)$ is true. To show that $P(t)$ is true. Assume that the numbers are sorted by their low-order $t - 1$ digits. Sort on digit t Two numbers that differ in digit t are correctly sorted. Two numbers are equal in digit t, they are put in the same order as the input $\implies$ correct order.

Randomized Algorithms

Average case: Expected running time when the input is chosen uniformly at random from the set of all possible $n!$ permutations.

Average Case Analysis of Quicksort

Suppose $A(n)$ is the average running time for quick sort of size n.

$$A(n) = \frac{1}{n!}\sum_{\pi} Q(\pi)$$

where $Q(\pi)$ is the time complexity when the input is permutation π. Let $P(i)$ be the set of permutations of $\{e_1, e_2, \cdots, e_n\}$ that begin with $e_i \implies P(i)$ constitutes $1/n$ of all possible permutations.
Let $G(n,i)$ be the average time of Quicksort over $P(i)$.

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

We can simplify the expression as follows:

$$nA(n) = [2\sum_{i=1}^{n} A(i-1)] + n(n-1) \quad (1)$$
$$(n-1)A(n-1) = [2\sum_{i=1}^{n-1} A(i-1)] + (n-1)(n-2) \quad (2)$$

Taking $(1)-(2)$, we get

$$\frac{A(n)}{n+1} - \frac{A(n-1)}{n} = \frac{2(n-1)}{n(n+1)} = \frac{4}{n+1} - \frac{2}{n}$$

We use telescoping,

$$\frac{A(n)}{n+1} - \frac{A(0)}{1} = \frac{4}{n+1} + \sum_{i=2}^{n}\frac{2}{i} - 2$$
$$\implies \frac{A(n)}{n+1} = \frac{4}{n+1} + \sum_{i=1}^{n}\frac{2}{i} - 4 \text{ (neat trick)}$$
$$\implies A(n) = 4 + (n+1)\sum_{i=1}^{n}\frac{2}{i} - 4(n+1)$$
$$\implies A(n) = 2(n+1)H(n) - 4(n) \text{ (harmonic series)}$$

Note that $H(n) = \log_e n + \gamma$ where $\gamma \approx 0.5772$

$$A(n) \approx 1.39n\log_2 n$$

As the value of n increases, the runtime of the algorithm tends towards the average.

Randomized Analysis of Quicksort

Probability that the run time of Randomized Quick Sort exceeds average by $x\% = n^{\frac{-x}{100}\ln\ln n}$
Define the $X_{i,j}$ to indicate whether $A*_i$ is compared to $A*_j$.
Lemma: For any $i<j$, we have

$$Pr[X_{i,j}] = E[X_{i,j}] = \frac{2}{j-i+1}$$

1. Probability depends upon the rank separation
2. Probability does not depends on the size of the array
3. Probability that e_i and e_{i+1} are compared = 1
4. Probability of comparison of e_0 and e_{n-1} is $\frac{2}{n}$

Expected number of comparisons

Let Y_{ij} for any $1 \le i < j \le n$, be a random variable defined as follows:

$$Y_{ij} = \begin{cases} 1 & \text{if } e_i \text{ is compared to } e_j \text{ during RQS of A} \\ 0 & otherwise \end{cases}$$

$$E[Y] = \sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{2}{j-i+1}$$
$$= 2\sum_{i=1}^{n}[\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-i+1}]$$
$$< 2\sum_{i=1}^{n}\left[1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right] - 2n$$
$$= 2nlog_e n - \Omega(n)$$

Balls into Bins

Expected number of empty bins

Let X_i be a random variable defined as follows:

$$X_i = \begin{cases} 1 & \text{if ith bin is empty} \\ 0 & \text{otherwise} \end{cases}$$

Note that $E[X_i] = 1 \cdot \text{Pr (ith bin is empty)} = (1-\frac{1}{n})^m$.
Using linearity of expectation, $E[X] = n(1-\frac{1}{n})^m$

Maximum Expected Number of Balls per Bin

Let X be $max\{X_i\}$. First form the following equation:

$$E[X] = 1P[X=1] + 2P[X=2] + \cdots + nP[X=n]$$
$$= P[X \ge 1] + P[X \ge 2] + \cdots + P[X \ge n]$$

Dig into a single term,

$$P[X \ge k] = P[X_1 \ge k \text{ or } X_2 \ge k \text{ or } \ldots \text{ or } X_n \ge k]$$

By Boole's inequality,

$$P[X \ge k] \le \sum P[X_i \ge k]$$

Given k balls, $Pr[\text{all k falls into the same bin}] = \frac{1}{n^k}$. Then, the probability that k balls fall into bin i, $n-k$ balls fall into other bins.

$$P[X_i = k] = \binom{n}{k}(\frac{1}{n})^k(1-\frac{1}{n})^{n-k}$$

Now, take all subsets of k balls. Let's name them $S_1, S_2, \ldots, S_n$. **Treat them as single elements each**.
Notice that $P[X_i \ge k] = P[\bigcup_i\{S_i \text{falls in bin } i\}]$. Applying the union bound,

$$P[X_i \ge k] \le \binom{n}{k}\frac{1}{n^k} \le (\frac{e}{k})^k$$

Note: we don't care about where the rest of the balls fall. Hence,

$$P[X \ge k] \le \sum_{i=1}^{n} P[X_i \ge k] = n(\frac{e}{k})^k$$
$$E[X] = 1P[X=1] + 2P[X=2] + \cdots + nP[X=n]$$
$$\le (k-1)\cdot P[X<k] + nP[X \ge k]$$
$$= (k-1)\cdot 1 + n \cdot n(\frac{e}{k})^k$$
$$\le k + n^2(\frac{e}{k})^k = E[X] = O(\frac{logn}{loglogn})$$

Expected number of collisions

$$E[X] = \sum_i E[X_i]$$
$$= \sum_i \binom{m}{2}\frac{1}{n^2}$$
$$= \frac{1}{n}\binom{m}{2}$$

Coupon Collector

Let T_i be the number of trials needed to collect a new coupon after $i-1$ distinct coupons have been collected.

Let T be the total number of trials to collect all n coupons.

$$E[T] = E[T_1] + \cdots + E[T_n]$$
$$= \sum_{i=1}^{n}\frac{n}{n-(i-1)}$$
$$= n\left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}\right)$$
$$= \theta(n\lg n)$$

Modified Coupon Collector

Let X be the total number of accesses. Let X_j be the number of accesses to obtain the $j-th$ coupon.

Hence, X_j is a geometric distribution with success probability $p_j = \frac{m-j+1}{n}$.

$$E[X] = \sum_{j=1}^{q}\frac{n}{m-j+1}$$
$$= \frac{n}{m} + \frac{n}{m-1} + \cdots + \frac{n}{m-q+1}$$

When $q<m$,

$$E[X] = \sum_{j=1}^{q}\frac{n}{m-j+1}$$
$$= n(H_m) - n(H_{m-q})$$
$$= \theta(n\lg\frac{m}{m-q})$$

Medians of sorted arrays

If we have two sorted arrays of size m and n respectively

- We can get the median in $O(m+n)$ time (merge them)
- Suppose $m=n$, then we can get the median in $\lg n$ time. First compute the median for both arrays in $O(1)$ time, then compare them. If $m1 = m2$, then return the median. WLOG, $m1 > m2$, then we remove the second half of array A and first half of array B and, recursively compute for the remaining.
- Suppose $m>n$, we can safely remove elements from $A[0:\frac{m-n}{2}]$ and $A[\frac{m+n}{2}:m-1]$ since these elements cannot possibly be the median (larger/smaller than $\frac{m+n}{2}$ elements — think of worst case). Then we use the same algorithm to compute the median.

Examples

- Frievald for Matrix Multiplication: $O(kn^2)$ (Monte Carlo)
- Smallest Enclosing Circle: $O(n)$ (Las Vegas)
- Minimum Cut: $O(mn)$ vs $O(m\log n)$ (Monte Carlo)
- Primality Testing: $O(n^6)$ vs $O(kn^2)$ (Monte Carlo)

Amortized Analysis

Given a sequence of c operations, total amortized cost is given by $f(n)*c \ge c*$ actual cost but this does not necessary mean that the amortized cost, $f(n) \ge C_a$.

Aggregate Analysis

Total time / total # of operations. Lacks precision and may not work for some cases. Useful for single operation.

Accounting Method

Impose extra charges on **inexpensive** operations and use it to pay for **expensive** operations. Any amount that is not used is stored in the bank for use by subsequenet operations. The bank balance must not go **negative**.

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

Potential Method

Denote $\phi(i)$ to be the potential at the end of ith operation.

$$\phi(0) = 0$$
$$\phi(i) \ge 0 \text{ for all } i$$

Notes

- Amortized analysis guarantees the **average performance of each operation** in the worst case.
- If we want to show that the actual cost of n operations is $O(g(n))$, it suffices to show that the amortized cost is $O(g(n))$.

Dynamic Programming

Cut-and-paste proof

1. Suppose your optimal solution is made using suboptimal solutions to subproblems
2. Show that if you were to replace the suboptimal subproblem solutions with optimal solutions, you would improve your optimal solution.
3. Hence assumption is false, and the optimal solution is indeed made using optimal subproblem solutions.

Longest Common Subsequence

$$L(n,m) = \begin{cases} 1 + L(n-1, m-1) & \text{if } x_n = y_m \\ \max(L(n-1,m), L(n,m-1)) & \text{otherwise} \end{cases}$$

Time: $O(nm)$. **Space**: $O(min(n,m))$.

Knapsack

$$K(n,m) = \begin{cases} v_n + K(n-1, m-w_n) & \text{if } w_n \leq m \\ K(n-1,m) & \text{otherwise} \end{cases}$$

Psuedocode (1D DP):

```
dp = [0] * (m+1)
for i in  1..n
  for j in m..0
    if w[i] <= j
      dp[j] = max(dp[j], dp[j-w[i]] + v[i])

return dp[m]
```

Time: $O(nW)$. **Space**: $O(W)$

- **not** a polynomial time algorithm since W can be represented in $O(\lg W)$ bits

Cutting rods

Uses the natural ordering of the subproblems: a subproblem of size i is "smaller" than a subproblem of size j if $i < j$. Thus, the procedure solves subproblems of sizes $j = 0, 1, \ldots, n$, in that order.

```
# S denotes the optimal first piece
# size for a rod of length n
r, s = [0] * (n+1), [0] * (n+1)
for j in range(1, n+1):
  q = -1
  for i in range(1, j+1):
    if q < p[i] + r[j-i]:
      q = p[i] + r[j-i]
      s[j] = i
  r[j] = q
return (r, s)
```

Matrix chain multiplication

Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix A_i. Let $s[i,j]$ be the index of the matrix which achieves an optimal parenthesization of $A_{i \ldots j}$.

```
n = len(p) - 1
# set m[1..n, 1..n] and s[1..n-1, 2..n]
for i in range(1, n+1):
  m[i][i] = 0 # trivial
for l in range(2, n+1): # chain length
  for i in range(1, n-l+2): # starting index
    j = i + l - 1 # ending index
    m[i][j] = -1
    for k in range(i, j): # partition index
      q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j
          ]
      if q < m[i][j]:
        m[i][j] = q
        s[i][j] = k
return (m, s)
```

Coin denominations

$$M[j] = \begin{cases} 0 & \text{if } j = 0 \\ \min_{i=1}^{k} M(j - d_i) + 1 & \text{otherwise} \end{cases}$$

Let $M[j]$ denotes the minimum number of coins required to change j cents. Let the coins be $d_1, d_2, \ldots, d_k$. Suppose $M[j] = t$ is optimal i.e. $j = \bigcup_i d_i$ for some $i \in \{i, \ldots, k\}$.

- Consider subproblem j', where $j' = j - d_i$ and $M[j'] = t - 1$.
- If this were suboptimal. then $M[j'] < t - 1$
- By cut-and-paste argument, we just need to add coin d_i to subproblem j' to reach subproblem j, then $M[j] = M[j'] + 1 < t - 1 + 1 = t$
- Contradicts claim that $M[j] = t$ is optimal.

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

Polygon triangulation

Overlapping subproblem:

$$T(n) = \sum_{i=2}^{n-1} T(k) + T(n+1-k) + d$$

$$T(n-1) = \sum_{i=2}^{n-2} T(k) + T(n-k) + d$$

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

$$T(n) = 3^{O(n)} = 2^{O(n)}$$

Psuedocode (Bottom up):

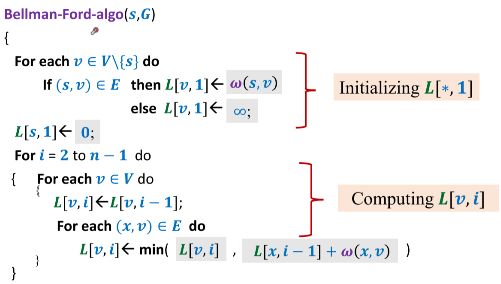
```
for d in range(2, n):
  for r in range(d+1, n):
    l = r - d
    for k in range(l+1, r):
      dp[l][r] = min(dp[l][r], dp[l][k] + dp[
          k][r] + c(l, k, r))
```

Psuedocode (Top down):

```
if (j - i == 1): return 0
# Not previously computed
if (memo[i][j] == -1):
  for k in range(i + 1, j):
    memo[i][j] = min(memo[i][j],
      t(i, k) + t(k, j) + c(i,k,j))
return memo[i][j]
```

Time Complexity: $O(N^3)$ Space Complexity: $O(N^2)$

Bellman Ford



Greedy

Fractional Knapsack

Runs in polynomial time, $O(npoly(\log n, \log W, \log M))$

- **Optimal substructure Property**
  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.
  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!
- **Greedy Choice Property**
  Let $j*$ be the item with the highest value per unit weight. Suppose an optimal knapsack containing x_1 kgs of item 1, x_2 kgs of item 2, ..., x_n kgs of item n such that

  $$x_1 + x_2 + \cdots + x_n = min(w_{j*}, W)$$

  We can replace this sum with $min(w_{j*}, W)$ kgs of item $j*$ and still have an optimal knapsack. Total value cannot decrease as $v_{j*} \geq v_i$ for all i.

Min. # of Days to travel up Hill, $H(n)$

Proof.Let the optimal solution, S be $s_1, s_2, \ldots, s_j, s_p, s_k$.
Let the greedy solution, S^* be $g_1, g_2, \ldots, g_j, g_p, g_k$.
Let j be the smallest index s.t. $s_j != g_j$. Since S does not make use of the greedy choice, we know that
$g_p = s_j + L \geq s_p \implies g_p + L \geq s_p + L \implies S^* \geq S$.
This does not make the solution any worse. Hence, S^* is also optimal. Runs in $O(n)$.

Longest Increasing Subsequence

DP solution: $O(n^2)$. Greedy solution: $O(n \log n)$

- Any increasing subsequence contains at most one item from each pile (**Weak Duality**)
- Length of LIS is at least the number of piles (**Strong Duality**)

Reductions

Polynomial-time reduction

$A \leq_p B$ if there is a polynomial-time reduction from A to B for some poly function $p(n)$.

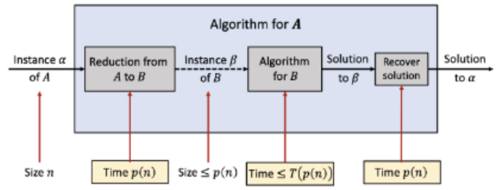
- Only shows that some instances of B are as hard as A. This does not mean that all instances of B are hard.
- Note that if A can be solved in poly time, then there is a reduction from A to **any** problem B. The reduction can just solve the instance for A by itself in polynomial time, and then generate an instance of B accordingly.

Running time

If there is a $p(n) - time$ reduction from problem A to problem B, and a $T(n)$-time algorithm for problem B on instance of size n then there is a

$$T(p(n)) + O(p(n))$$

time algorithm to solve problem A on instances of size n.



Psuedo-polynomial algorithms

An algorithm runs in pseudo-polynomial time if

- polynomial in the numeric value of the input, but
- exponential in the length of the representation of input

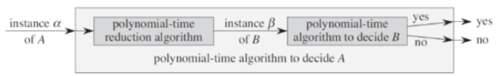
Intractability

Decision reduces to optimization.Given an instance of the optimization problem and a number k, we can ask whether there is a solution with value $\leq k$.

(**Cook reduction**) Solve an instance of A by making multiple calls to the subroutine for B. If the reduction algorithm is poly-time, then we call it a poly-time Cook reduction. (The number of calls to the subroutine of B is also bounded by a polynomial if total reduction time is bounded by some polynomial.)

(**Karp's Reduction**) Given **two decision problems A and B**, a polynomial-time reduction from A to B is a transformation from instances α of A to instances 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



Independent Set

3-SAT $\leq_p$ Independent Set. Construct a graph where any 3 literals in clause forms a triangle and connect literals to its negation. Set k = number of clauses. Note that the same literal in different clauses are considered different vertices. If there exists a satisfying assignment of the 3-SAT instance, then there exists an independent set of size k in the graph.

Vertex Cover

To check whether G has a vertex cover of size k, we can check whether G has an independent set of size $n - k$.
There exists an optimal greedy solution for trees.

Set Cover

Given a set of elements S and a collection of subsets F of S, a set cover is a subset of F such that the union of all elements in the subset covers S.
Vertex cover $\leq_p$ Set cover. Number each edge with a unique number. For each vertex, create a set that contains all the edges incident to the vertex.

Circuit Satisfiability Problem, CNF-SAT, 3SAT

A DAG with nodes corresponding to AND, NOT, OR gates and n binary inputs, does there exist any binary input which gives output 1?
(**Cook & Levin**) CSP is NP-Complete

$$CSP \leq_p CNF - SAT \leq_p 3SAT$$

Hamiltonian Cycle and Path

- $HC \leq_p TSP$. Make the original edges in G to cost and make the edges that isn't in G originally to cost more. It can be implemented in $O(|V|^2 + |E|)$ time.
- $HC \leq_p HP$. Take **any** vertex $v \in V$, and create a copy v' of it. For each edge (u, v) in E, add the edge (u, v') to E'. Further, create two new vertices, s and t and connect them to v and v' respectively. Key insight: s and t are vertices of degree one, so any HP must start and end at either s or t.
- $HP \leq_p$ DEGREE-BOUNDED SPANNING TREE. For each $v \in V$, add three new vertices v_1, v_2, v_3 and connect them to v with edges of weight 1. Key insight: The three added vertices guarantees that the HP must visit v exactly once.

Finding Family

MAX-CLIQUE $\leq_p$ FINDING-FAMILY. Set the edge between two vertices as a node in R. If there exists a clique of size k, then there exists a family of size k.

Hitting set

Given a ground set X of elements and also a grouping collection C of subsets available in X and an integer k, the task is to find the smallest subset, H, of X, such that it hits every set comprised in C.

3-SAT $\leq_p$ HITTING-SET. Set each clause as a set in C. Additionally, for each literal, set the set containing the literal and its negation as a set in C. If there exists a satisfying assignment, then there exists a hitting set of size k. Note that the value of k is the number of literals.

VERTEX COVER $\leq_p$ HITTING-SET. Set the endpoints of each edge $\in E$ to be a set in C.

Subset sum $\leq_p$ Partition

Let (L, B) be an instance of subset sum, where L is a list (multiset) of numbers, and B is the target sum. Let L' be the list formed by adding $S + B$,$2S - B$ to L.

1. If there is a sublist $M \subset L$ summing to B, then L' can be partitioned into two equal parts: $M \cup 2S - B$ and $L - M \cup \{S + B\}$. Indeed, the first part sums to $B + (2S - B) = 2S$, and the second to $(S - B) + (S + B) = 2S$.
2. If L' can be partitioned into two equal parts P1,P2, then there is a sublist of L summing to B. Indeed, since $(S + B) + (2S - B) = 3S$ and each part sums to 2S, the two elements belong to different parts. Without loss of generality, $2S - B \in P1$. The rest of the elements in P1 belong to L and sum to B.

Complexity

Non-Deterministic Polynomial (NP)

- Defined as the class of problems for which polynomial time verifiable certificates of YES-instance exist
- There is a verification algo $V(x, y)$ that takes in an instance X and a certificate y with $|v| = poly(|x|)$ such that

$$\exists y s.t. V(x, y) = 1 \iff x \text{ is a YES-instance}$$

- $P \subset NP$ because certificate can be anything, while verifier $V(x,)$ simply solve for instance x by itself and check if it is a YES-instance.

NP-Complete

- If you could come up with a polynomial-time algorithm for an NP-complete problem, you would show $P = NP$.

Useful Knowledge

Union Find

- any sequence of m union operations can involve at most $2m$ many elements.
- each element can be union at most $logn$ times.