

CS653 Analysis of Algorithms

1 Mathematical Foundations

1.1 Common functions

Reading: CLRS 3.2

- Ceilings and floors: $\lceil x \rceil$ and $\lfloor x \rfloor$.
- Modular arithmetic: $a \bmod n = a - \lfloor a/n \rfloor n$. $a \equiv b \bmod n$ iff $a \bmod n = b \bmod n$.
- Polynomials: $p(n) = \sum_{i=0}^d a_i n^i$. (Note: Coefficients a_i and degree d are constants.)
- Exponentials: $a^0 = 1$, $a^{-1} = \frac{1}{a}$, $a^m \cdot a^n = a^{m+n}$, $a^m / a^n = a^{m-n}$.
 $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$. (Note: $e = 2.71828\dots$)
 $\lim_{n \rightarrow \infty} (1 + \frac{x}{n})^n = e^x$.
- Logarithms: $\log n = \log_2 n$ or $\log_c n$ for some c we don't care about.
 $\log(ab) = \log a + \log b$, $\log(\frac{a}{b}) = \log a - \log b$.
 $\log_a b = \frac{\log_c b}{\log_c a}$.
 $\log_a b^n = n \log_a b \neq (\log_a b)^n$, $a^{\log_a n} = n$, $a^{\log_c b} = b^{\log_c a}$.
 $\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots$.
- Factorials: $n! = n \cdot (n-1) \cdots 2 \cdot 1$.
 $n! = n \cdot (n-1)!$, $0! = 1$.
Stirling's approximation: $n! = \sqrt{2\pi n} (\frac{n}{e})^n (1 + \Theta(\frac{1}{n}))$. (Note: Θ means having the same order of magnitude.) The following approximation also holds: $n! = \sqrt{2\pi n} (\frac{n}{e})^n e^{\alpha_n}$, where $\frac{1}{12n+1} < \alpha_n < \frac{1}{12n}$.
 $\log n! = \Theta(n \log n)$.
- Functional iteration: A function f applied iteratively i times to an initial argument n . Defined recursively, $f^{(0)}(n) = n$ and $f^{(i)}(n) = f(f^{(i-1)}(n))$ for $i > 0$. (Note: The distinction between $f^{(i)}(n)$ and $f^i(n)$.)
For example, if $f(n) = 2n$ then $f^{(i)}(n) = 2^i n$.
- The log star function: $\log^* n = \min\{i \geq 0 : \log^{(i)} n \leq 1\}$, which is a very slowly growing function. $\log^* 2 = 1$, $\log^* 4 = 2$, $\log^* 16 = 3$, $\log^* 65536 = 4$, $\log^* 2^{65536} = 5$.
- Combinatorics: Permutation $n!$ (order matters); Combination $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ (order doesn't matter).
Pascal's triangle: $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.
Binomial expansion: $(x+y)^n = \sum_{k=0}^n \binom{n}{k} x^k y^{n-k}$.

1.2 Sums

Reading: CLRS A

- Arithmetic: $1 + 2 + \dots + n = \frac{1}{2}n(n+1)$.
- Geometric: $1 + r + r^2 + \dots + r^n = \frac{r^{n+1}-1}{r-1}$ for $r \neq 1$.
 $1 + r + r^2 + \dots = \frac{1}{1-r}$ for $|r| < 1$.

- Harmonic: $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \ln n + \gamma + \frac{\varepsilon}{2n}$ for $\gamma = 0.5772156649 \dots$ (Euler's constant) and $0 < \varepsilon < 1$.

Example: Prove that $H_n < \ln n + 1$.

Proof: $H_n = \sum_{i=1}^n \frac{1}{i} = 1 + \sum_{i=2}^n \frac{1}{i} < 1 + \int_1^n \frac{1}{x} dx = 1 + \ln n$.

f1.pdf

Remark: Use integrals to bound summations: Assume $f(x)$ is monotonically decreasing, then $\int_m^{n+1} f(x) dx \leq \sum_{k=m}^n f(k) \leq \int_{m-1}^n f(x) dx$. (What if the function is monotonically increasing?)

- Binomial: $\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = 2^n$.

- Other useful sums:

$$\sum_{i=1}^n i^2 = \frac{1}{6}n(n+1)(2n+1).$$

$$\sum_{i=1}^n i^3 = \left(\sum_{i=1}^n i\right)^2.$$

$$\sum_{i=1}^n ix^{i-1} = \frac{nx^{n+1} - (n+1)x^n + 1}{(x-1)^2}.$$

1.3 Asymptotic notation

Reading: CLRS 3.1

- Used to compare the growth rate or order of magnitude of increasing functions. “Asymptotic” deals with the behavior of functions in the limit, for sufficiently large values of variables.
- $f(n) = O(g(n))$ if $\exists c, n_0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$.
- $f(n) = \Omega(g(n))$ if $\exists c, n_0$ such that $f(n) \geq cg(n)$ for $n \geq n_0$.
- $f(n) = \Theta(g(n))$ if $\exists c_1, c_2, n_0$ such that $c_1g(n) \leq f(n) \leq c_2g(n)$ for $n \geq n_0$.
- $f(n) = o(g(n))$ if $\forall c \exists n_0$ such that $f(n) < cg(n)$ for $n \geq n_0$.
- $f(n) = \omega(g(n))$ if $\forall c \exists n_0$ such that $f(n) > cg(n)$ for $n \geq n_0$.

- *Remarks:*

- In CLRS, the above notation is defined as sets of functions. For example, $f(n) \in O(g(n))$. But we will simply follow the mainstream here.
- $O(\leq)$, $\Omega(\geq)$, $\Theta(=)$, $o(<)$, $\omega(>)$.
- $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$, and $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$.
- $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
- $f(n) = O(g(n))$ if $f(n) = o(g(n))$, and $f(n) = \Omega(g(n))$ if $f(n) = \omega(g(n))$.
- An alternative definition for $f(n) = o(g(n))$ is $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$. Likewise, an alternative definition for $f(n) = \omega(g(n))$ is $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$.
- Asymptotic notation ignores constant factors and lower-order terms.
- *Rule of thumb:* constant \leq polylogarithmic \leq polynomial \leq exponential \leq superexponential.
Example: 1, $\sqrt{\log n}$, $\ln n$, $(\log n)^2$, \sqrt{n} , $\sqrt{n} \log n$, $10n$, $n \log n$, n^2 , $n^{\log \log n}$, 2^n , $n2^n$, $n!$, 2^{2^n} .
- Taking logarithms helps: $f(n) = O(g(n))$ iff $\log f(n) = O(\log g(n))$.

1.4 Proof techniques

- Proving by contradiction:

The following three statements are logically equivalent:

1. If A then B .
2. If not B then not A .
3. If A and not B then not C , where C is a proved fact or axiom.

Example: Use contradiction to prove that (a) There are infinitely many prime numbers; (b) There exist two irrational numbers x and y such that x^y is rational; and (c) $\sqrt{2}$ is irrational.

Proof: Exercises or references.

- Proving by induction:

The following statements are mathematically equivalent:

1. $P(n)$ for integers $n \geq c$.
2. Simple integer induction: $P(c)$ and $P(n-1) \rightarrow P(n)$. (What are inductive basis, inductive hypothesis, and inductive step?)
3. General integer induction: $P(c)$ and $(\forall i: c \leq i \leq n-1)P(i) \rightarrow P(n)$.

There are two additional types of induction:

1. Structural induction: To prove a property $P(X)$ of a recursively defined structure X , it is suffice to prove that $P(X)$ is true for the basis structure X and $P(Y_1) \wedge P(Y_2) \wedge \dots \wedge P(Y_k) \rightarrow P(X)$, where X is constructed from Y_1, Y_2, \dots, Y_k .
2. Mutual induction: Used in the case when a single statement $P_1(n)$ can not be proved by induction but rather $P_1(n)$ together with some other statements $P_2(n), \dots$ can be proved successfully by induction.

Example: Use induction to prove that (a) The tiling problem can always be solved; (b) Every positive composite integer can be expressed as a product of prime numbers; and (c) Every tree has one more node than it has edges.

Proof: Exercises or references.

Remark: What is the tiling problem? Given a board divided into $m = 2^k$ squares in each row and each column with one square marked special. Given a supply of tiles, each being a 2×2 board with one square removed. The puzzle is to cover the board with the tiles so that each square is covered exactly once, with the exception of the special square, which is not covered at all.

1.5 Solving recurrences

Reading: CLRS 4

- Recurrence is an equation or inequality that defines a function in terms of the function's values on smaller inputs. For example, $T(1) = \Theta(1)$ (boundary condition) and $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ for $n \geq 2$ or almost equivalently, $T(1) = 1$ and $T(n) = 2T(\frac{n}{2}) + n$ for $n \geq 2$.
- *Remark:* We may neglect some technical details due to our interest in asymptotic solutions:
 - Relax the integer argument requirement on functions.
 - Assume boundary condition $T(n) = \Theta(1)$ for small n if not defined.
 - Use $\Theta(f(n))$ or $f(n)$ at will in the recursive definition.

- The substitution method: Guess and verify.

Example: $T(n) = 2T(\frac{n}{2}) + n$.

First guess $T(n) = O(n \log n)$. Next prove that $T(n) \leq cn \log n$ for some c and $n \geq 2$. (Why not start from $n = 1$?)

Inductive basis: For $n = 2$, $T(2) = 2T(1) + 2 = 4 \leq c2 \log 2$ if $c \geq 2$.

Inductive step: Assume $T(\frac{n}{2}) \leq c \frac{n}{2} \log \frac{n}{2}$.

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + n \\ &\leq cn \log \frac{n}{2} + n \\ &= cn \log n - cn + n \\ &< cn \log n. \end{aligned}$$

Remarks:

- Making a good guess.
- To prove $T(n) = O(f(n))$, sometimes we use an inequality stronger than $T(n) \leq cf(n)$ in the induction (such as $T(n) \leq 20cf(n)$).
- Avoid using asymptotic notation in the inductive proof.

Exercise: $T(n) = T(n-1) + n$. What is wrong with the following proof?

First guess $T(n) = O(n)$.

Inductive basis: For $n = 1$, $T(1) = 1 = O(1)$.

Inductive step: Assume $T(n-1) = O(n-1)$

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= O(n-1) + n \\ &= O(n). \end{aligned}$$

- Changing variables.

Example: $T(n) = 2T(\sqrt{n}) + \log n$.

First, renaming $m = \log n$ yields $T(2^m) = 2T(2^{m/2}) + m$. Next, renaming $S(m) = T(2^m)$ yields $S(m) = 2S(m/2) + m$. Since $S(m) = O(m \log m)$, $T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n \log \log n)$.

- The iteration method: Apply recurrence until a summation pattern can be figured out.

Example: $T(n) = 3T(\frac{n}{4}) + n$.

Assume $n = 4^k$, so $k = \log_4 n$.

$$\begin{aligned} T(n) &= 3T(\frac{n}{4}) + n \\ &= 3^2 T(\frac{n}{4^2}) + \frac{3}{4}n + n \\ &= 3^3 T(\frac{n}{4^3}) + (\frac{3}{4})^2 n + (\frac{3}{4})n + n \\ &= \dots \\ &= 3^k T(\frac{n}{4^k}) + ((\frac{3}{4})^{k-1} + \dots + (\frac{3}{4}) + 1)n \\ &= 3^{\log_4 n} + \frac{1 - (\frac{3}{4})^{\log_4 n}}{1 - \frac{3}{4}}n \\ &= 4n - 3 \cdot 3^{\log_4 n} \\ &= O(n). \end{aligned}$$

Exercise: Solve $T(n) = \sqrt{n}T(\sqrt{n}) + n$ by iteration.

- The recursion-tree method: Similar to the iteration method, use a tree for bookkeeping.

Example: $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + n$.

f2.pdf

Exercise: Solve $T(n) = T(\alpha n) + T((1 - \alpha)n) + n$, where $0 < \alpha < 1$, by recursion tree.

- The master method:

Theorem: If $T(n) = aT(\frac{n}{b}) + f(n)$ for $a \geq 1$ and $b > 1$, then

- (a) if $f(n) = O(n^{(\log_b a) - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$;
- (b) if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$;
- (c) if $f(n) = \Omega(n^{(\log_b a) + \epsilon})$ for $\epsilon > 0$ and if $af(\frac{n}{b}) \leq cf(n)$ for $c < 1$ and all large n , then $T(n) = \Theta(f(n))$.

Remark: The master method does not cover all cases.

Example: $T(n) = 3T(\frac{n}{4}) + n \log n$.

$a = 3$, $b = 4$, and $f(n) = n \log n$. Case (c) applies. So $T(n) = \Theta(n \log n)$.

Exercise: Solve $T(n) = 4T(\frac{n}{2}) + f(n)$ by the master theorem for $f(n) = n, n^2, n^3$.

2 Analysis of Algorithms

2.1 An overview

Reading: CLRS 1

- Problem types:
By difficulty: conceptually hard, analytically hard, computationally hard, computationally unsolvable.
- Algorithms:
Correctness, finiteness, definiteness (unambiguity), effectiveness, output.
Pseudocode = human language + math language + computer language.
Analysis of algorithms: w/o programming, testing, and debugging.
- Complexity measures: Time, space \leftrightarrow worst, average.
Time is measured as the number of basic steps in a prespecified computation model.
- Models of computation (to define basic steps):
Turing machine: Basic step: Read-write-move-change.
Random Access Machine: Basic step: LOAD, STORE, ADD, SUB, MULT, DIV, READ, WRITE, JUMP, JGTZ, JZERO, HALT.
Pseudocode: Basic step: +, −, *, /, assignment, compare, if, i/o. (for, while, ** are not.)
Polynomial equivalence of all reasonable computing models.
- Instance: A specific input.
Example: Problem: Sort a list of numbers into increasing order; Instance: $L = (5, 1, 4, 2, 3)$.
- Instance (Input) size: The amount of space used to store the instance.
Example:

Instance	Size
List	# of items
Matrix	# of rows and columns
Graph	# of vertices and edges
Integer	# of bits

- Time complexity: $T(n)$: # of basic steps in the pseudocode model.
Remarks:
 - Functions of input size, not input value, represented in O , Ω , Θ (easy to compare; asymptotic performance for large input size).
 - Input-value independent, programming independent, language independent, machine independent.
- A spectrum of complexity:
Classify problems into categories according to the worst-case time complexity of their fastest possible (optimal) algorithms.

Degree of difficulty	Problem examples
Unsolvable	Halting problem
Superexponential	Presburger arithmetic 2^{2^n}
Exponential	Circularity attribute grammar 2^n
Polynomial	Linear programming (LP)
n^c , for $2 \leq c < 3$	Matrix multiplication
$n \log n$	Sorting by comparison
n	Selecting the k th largest

2.2 Worst-case analysis

Reading: CLRS 2.1 and 2.2

- Definition:

I_n : Any instance of size n ;

$t(I_n)$: Time (# of basic steps) spent on I_n by the algorithm;

$T(n)$: Worst-case time complexity of any instance of size n , $T(n) = \max_{I_n} \{t(I_n)\}$.

- Insertion sort: $A \Rightarrow A$ sorted in increasing order.

Insertion.Sort(A)

for $j \leftarrow 2$ to n

$key \leftarrow A[j]$

$i \leftarrow j - 1$

 while $i > 0$ and $A[i] > key$

$A[i + 1] \leftarrow A[i]$

$i \leftarrow i - 1$

$A[i + 1] \leftarrow key$

To insert $A[j]$ into the sorted $A[1 \dots j - 1]$, the algorithm will make at most $j - 1$ comparisons and $j - 1$ shifts. So the overall time complexity, which is dominated by the number of comparisons and the number of shifts, is at most $2 \sum_{j=2}^n (j - 1) = \Theta(n^2)$.

- Euclid's algorithm (300 B.C.): Greatest common divisor.

If $n \neq 0$, then $\gcd(m, n) = \gcd(n, m \bmod n)$;

If $n = 0$, then $\gcd(m, n) = m$.

Euclid(m, n) // Assume $m \geq n$

 while $n > 0$

$t \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow t$

 return m

The time complexity of the algorithm is determined by the number of iterations of the while-loop. Let it be k . Also let m_i and n_i be the values of m and n at the end of the i th iteration, respectively.

We observe that (1) $n_k = 0$, $n_i \geq 1$ for $i < k$, and $n_0 > n_1 > \dots > n_k$; (2) $m_i = n_{i-1}$, which implies that $m_i \geq n_i$ for $i \geq 1$; (3) $n_i = m_{i-1} \bmod n_{i-1} < \frac{m_{i-1}}{2} = \frac{n_{i-2}}{2}$ for $i \geq 2$. (Note: If $a \geq b$, then $a \bmod b < \frac{a}{2}$.)

We then have $1 \leq n_{k-1} < \frac{n_{k-3}}{2} < \frac{n_{k-5}}{2^2} < \dots < \frac{(n_1, n_0)}{2^{\lfloor k/2 \rfloor - 1}} \leq \frac{n_0}{2^{\lfloor k/2 \rfloor - 1}}$.

Thus, $\frac{k}{2} - 1 \leq \lceil \frac{k}{2} \rceil - 1 < \log n_0$, which implies $k \leq 2 + 2 \log n_0$.

So $T(m, n) = O(k) = O(\log n)$.

- Multiplying two integers x and y :

Assume that x (multiplicand) and y (multiplier) have the same number of figures, $n = 2^k$, denoted $(x)_n$ and $(y)_n$. Then $(x)_n = 10^{\frac{n}{2}} \cdot (a)_{\frac{n}{2}} + (b)_{\frac{n}{2}}$ and $(y)_n = 10^{\frac{n}{2}} \cdot (c)_{\frac{n}{2}} + (d)_{\frac{n}{2}}$.

So $(x)_n \cdot (y)_n = 10^n ((a)_{\frac{n}{2}} \cdot (c)_{\frac{n}{2}}) + 10^{\frac{n}{2}} ((a)_{\frac{n}{2}} \cdot (d)_{\frac{n}{2}} + (b)_{\frac{n}{2}} \cdot (c)_{\frac{n}{2}}) + (b)_{\frac{n}{2}} \cdot (d)_{\frac{n}{2}}$. The problem of multiplying two integers of size n is now reduced to four subproblems of multiplying two integers of size $\frac{n}{2}$.

$T(n) = 4T(\frac{n}{2}) + n = \Theta(n^2)$ by the master theorem.

Now let $s = (a)_{\frac{n}{2}} \cdot (c)_{\frac{n}{2}}$, $t = (b)_{\frac{n}{2}} \cdot (d)_{\frac{n}{2}}$, and $r = ((a)_{\frac{n}{2}} + (b)_{\frac{n}{2}}) \cdot ((c)_{\frac{n}{2}} + (d)_{\frac{n}{2}})$. Then $(x)_n \cdot (y)_n = 10^n s + 10^{\frac{n}{2}} (r - s - t) + t$. Only three subproblems have to be solved in the recursion.

$T(n) = 3T(\frac{n}{2}) + n = \Theta(n^{\log_2 3}) = \Theta(n^{1.585})$ by the master theorem.

2.3 Average-case analysis

Reading: CLRS 2.2

- Definition:

I_n : Any instance of size n ;

$t(I_n)$: Time (# of basic steps) spent on I_n by the algorithm;

$p(I_n)$: Probability that I_n appears as input;

$T(n)$: Average-case time complexity of any instance of size n , $T(n) = \sum_{\forall I_n} p(I_n)t(I_n)$.

- Insertion sort:

Assumptions: (1) Distinct items; (2) Each permutation with equal probability to occur.

$T(n)$ = average # of pairwise comparisons = $\sum_{j=2}^n c_j$, where c_j is the average # of comparisons to insert $A[j]$ (key) into the sorted $A[1 \dots j-1]$. For key , there are j possible positions, with probability $\frac{1}{j}$ for each.

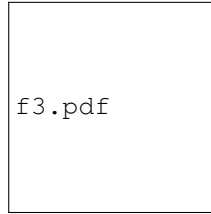
So $c_j = \frac{1}{j} \cdot 1 + \frac{1}{j} \cdot 2 + \dots + \frac{1}{j} \cdot (j-1) + \frac{1}{j} \cdot (j-1) = \frac{1}{2}(j+1) - \frac{1}{j}$.

Therefore, $T(n) = \sum_{j=2}^n (\frac{1}{2}(j+1) - \frac{1}{j}) = \frac{1}{4}n^2 + \frac{3}{4}n - H_n = \Theta(n^2)$.

- Binary search

Given $A[1 \dots n]$ sorted and x (a query). Is x a member of A ?

Example: Let $n = 7$. The binary search algorithm can be illustrated by the following decision tree.



Assumptions: (1) $x \in A$; (2) Distinct items in A ; (3) $Pr(x = A[i]) = \frac{1}{n}$ for any $i = 1, 2, \dots, n$; (4) $n = 2^k - 1$ for some k .

Binary search \Rightarrow Decision tree of n nodes (full binary tree with k levels).

Average-case time \Rightarrow Average length of a path from the root to a tree node.

Level	# of nodes	average length
1	1	$\frac{1}{n} \cdot 1$
2	2	$\frac{2}{n} \cdot 2$
3	4	$\frac{4}{n} \cdot 3$
...
k	2^{k-1}	$\frac{2^{k-1}}{n} \cdot k$

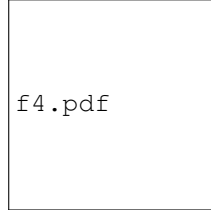
$$\begin{aligned}
 T(n) &= \sum_{i=1}^k \frac{2^{i-1}}{n} \cdot i \\
 &= \frac{1}{n} \sum_{i=1}^k i \cdot 2^{i-1} \\
 &= \frac{1}{n} (k \cdot 2^k - 2^k + 1) \\
 &= k + \frac{k}{n} - 1 \\
 &= \Theta(\log n)
 \end{aligned}$$

- Binary search trees (BST)

What is the average number of comparisons needed to insert n distinct random elements into an initially empty BST?

$$T(0) = T(1) = 0.$$

Assume that $A = (a_1, \dots, a_n)$ is the list given and that $B = (b_1, \dots, b_n)$ is A sorted increasingly. That A is a random sequence implies that a_1 is equally likely to be b_j for any $1 \leq j \leq n$. Consider the tree obtained after the insertion of all n numbers.



So $T(n) = \frac{1}{n} \sum_{j=1}^n (n-1 + T(j-1) + T(n-j)) = n-1 + \frac{2}{n} \sum_{j=0}^{n-1} T(j)$ for $n \geq 2$.

To solve this recurrence, we guess $T(n) = O(n \log n)$, i.e., $T(n) \leq cn \ln n$ for some c . We prove by induction that this is indeed the case.

When $n = 1$, $T(n) = 0 \leq c \cdot 1 \cdot \ln 1$ for any c . So the basis holds. Assume that when $j \leq n-1$, $T(j) \leq cj \ln j$. Now consider n .

$$\begin{aligned} T(n) &= n-1 + \frac{2}{n} \sum_{j=0}^{n-1} T(j) \\ &\leq n-1 + \frac{2}{n} \sum_{j=1}^{n-1} cj \ln j \\ &\leq n-1 + \frac{2}{n} c \int_2^n x \ln x dx \\ &\leq n-1 + \frac{2}{n} c \left(\frac{1}{2} n^2 \ln n - \frac{1}{4} n^2 \right) \\ &= n-1 + cn \ln n - \frac{c}{2} n \\ &\leq cn \ln n \quad \text{for } c \geq 2 \end{aligned}$$

2.4 Amortized analysis

Reading: CLRS 17.1–17.3

- Amortization:

- (1) To put money aside at intervals, as a sinking fund, for gradual payment of a debt;
- (2) To average the running times of operations in a sequence over the entire sequence.

- Motivation:

A sequence of data structure related operations, rather than just a single operation, is performed. (An operation may change the data structure, thus affect the next operation.)

When is the total time complexity of the entire sequence?

Worst-case analysis: sum of worst-case time of each operation (which may never be achieved).

Average-case analysis: average over all possible inputs, inaccurate probabilistic assumptions, hard.

Amortized analysis: average over successive operations, average the time per operation over a worst-case sequence. It guarantees the average performance of each operation in the worst case.

- Stack manipulation: an example.

Unit-time primitives: push and pop.

An operation contains zero or more pop followed by a push.

A sequence contains m operations defined above. For instance, (push), (pop, push), (push), Assume that any given sequence never pops when the stack is empty.

Initially the stack is empty.

Question: What is the total time/cost of a sequence of m operations? (as a function of m)

Worst-case analysis: m^2

Amortized analysis: $2m$ (Each operation causes one push and possibly one later pop.)

- A banker's view of amortization:

To represent prepaid work as credit stored with specific objects within the data structure.

Coin-operated computer: 1 credit \Rightarrow 1 time unit \Rightarrow 1 primitive.

Sequence of operations: O_1, \dots, O_m , where operation O_i is allocated with credits c_i .

Assumptions: (1) Unused credits are carried over to later operations; (2) Operations can borrow credits as long as any debt is paid off eventually.

If all sequences of length m can be performed with the allocated credits, then the total time is no larger than $\sum_{i=1}^m c_i$.

Keys: (1) How to pick the smallest possible c_i ; (2) How to prove all sequences can be performed with the allocated credits c_1, \dots, c_m .

Consider the stack manipulation example. Let $c_i = 2$ for $i = 1, \dots, m$. For any operation, one credit is used to execute push in the operation, while the other is saved for pop (of the same element) in a later operation. So in other words, each push is paid right away and each pop is paid for by a saved credit. So the total time is bounded from above by $\sum_{i=1}^m c_i = 2m$.

- A physicist's view of amortization:

To represent prepaid work as potential energy that can be released for future operations. The potential is associated with the data structure as a whole rather than with specific objects within the data structure.

Define a potential function $\Phi : D \rightarrow R$. Assume that an operation takes t_i time units to change the data structure from D_{i-1} to D_i . Then the amortized time of the operation is

$$a_i = t_i + \Phi(D_i) - \Phi(D_{i-1}),$$

and the total time for the sequence is

$$\begin{aligned}
\sum_{i=1}^m t_i &= \sum_{i=1}^m (a_i - \Phi(D_i) + \Phi(D_{i-1})) \\
&= \Phi(D_0) - \Phi(D_m) + \sum_{i=1}^m a_i \\
&\leq \sum_{i=1}^m a_i \quad \text{if } \Phi(D_0) \leq \Phi(D_m).
\end{aligned}$$

Key: How to choose Φ .

Consider the stack manipulation example. The underlining data structure is the stack. Define $\Phi(D)$ = # of items in the stack. Since $\Phi(D_0) = 0$ and $\Phi(D_m) \geq 0$, then $\Phi(D_0) \leq \Phi(D_m)$. If O_i has k pops and 1 push, and D contains j items before O_i is performed, then $\Phi(D_{i-1}) = j$ and $\Phi(D_i) = j - k + 1$. So $a_i = t_i + \Phi(D_i) - \Phi(D_{i-1}) = (k + 1) + (j - k + 1) - j = 2$. Therefore, the total time is $\sum_{i=1}^m t_i \leq \sum_{i=1}^m a_i = 2m$.

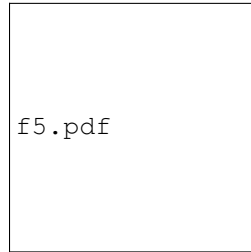
2.5 Disjoint sets

Reading: CLRS 21

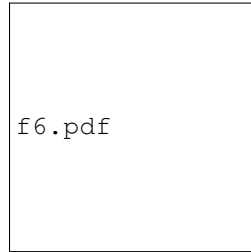
- Purposes: (1) Use good data structure to achieve algorithm efficiency; (2) Use amortization (it's accounting method) to give tight estimation of algorithm complexity.
- Problem: Initially, we are given n singletons: $S_i = \{i\}$ for $i = 1, \dots, n$. We wish to execute a sequence of operations of the following two types: (1) $\text{union}(S_i, S_j)$ returns $S_i \cup S_j$, where S_i and S_j are disjoint; (2) $\text{find}(i)$ returns the name of the set containing i . How can we organize the data (sets) such that any sequence of intermixed unions and finds can be performed efficiently?
- Data structure: Set \Rightarrow tree (arbitrary); Set name \Rightarrow root; Sets \Rightarrow forest (use parent array).

Initially, there are n singletons, corresponding to n single-node trees in a forest, which is represented by a parent array of size n with 0 in each entry. ($\text{parent}[i]$ gives the parent of i in the forest. If $\text{parent}[i] = 0$, i is a root.)

Implementation of union (by size): $\text{union}(i, j)$ in $O(1)$.



Implementation of find (by path compression): $\text{find}(i)$ in $O(d)$, where d is the depth of i in the tree.



- Complexity: Given an intermixed sequence σ of q unions and p finds. What is the worst-case time complexity for executing σ on $\{1\}, \dots, \{n\}$?

Note: The worst-case analysis gives an overly pessimistic bound of $O(q + pn)$ or $O(q + p \log q)$ (by P3 later). We will see next how a tighter bound can be obtained by an amortized analysis.

Let σ' be σ with all finds removed. Let $\text{Forest}(\sigma')$ be the forest resulting from executing σ' .

Rank $r(i)$: Height (# of nodes on the longest path from i to a leaf) of the subtree rooted at i in the forest resulting from executing σ' .

Properties:

P1: Any tree has no larger than $q + 1$ nodes in $\text{Forest}(\sigma')$. (Branches).

P2: Node i has at least $2^{r(i)-1}$ descendants (including i) in $\text{Forest}(\sigma')$. (Induct on $r(i)$). Consider the subtree rooted at node i , which must be constructed by some union operations. There must be a union operation that merges a smaller-size tree with height $r(i) - 1$ to a tree rooted at i . Thus the number of nodes in the tree obtained after the union is at least $2 \cdot 2^{r(i)-2} = 2^{r(i)-1}$.

P3: $r(i) \leq \lfloor \log(q + 1) \rfloor + 1$, for any i . (By contradiction, assume $r(i) \geq \lfloor \log(q + 1) \rfloor + 2$. By P2, the number of descendant is at least $2^{\lfloor \log(q+1) \rfloor + 1} > 2^{\log(q+1)} = q + 1$. A contradiction to P1.)

P4: There are at most $\frac{n}{2^{r-1}}$ nodes of rank r . (By contradiction, assume for some rank r there are more than $\frac{n}{2^{r-1}}$ nodes with rank r . By P2 each node has at least 2^{r-1} descendants. The number of nodes in these (disjoint) subtrees is more than $\frac{n}{2^{r-1}} \cdot 2^{r-1} = n$. A contradiction!)

P5: At any time during the execution of σ , the ranks of the nodes on a path from a leaf to a root increase strict monotonically. (For nodes w and v , if w is a proper descendant of v at some time of executing σ , then they must still have the same descendant-ancestor relation in $Forest(\sigma')$. So $r(w) < r(v)$.)

Definitions: $F(0) = 1$ and $F(n) = 2^{F(n-1)}$ for $n \geq 1$. Let G be the inverse of F , i.e., $G(n) = \min\{k \geq 0 | F(k) \geq n\}$. Note that $G(F(n)) = n$ and $G(n) = \log^* n$.

n	$F(n)$	$G(n)$
0	1	0
1	2	0
2	4	1
3	16	2
4	65536	2
5	2^{65536}	3
...
16	...	3
17	...	4
...

We next partition the ranks into groups by putting rank r into group $G(r)$. Thus, group 0 contains rank 1; group 1 contains just rank 2; group 2 contains ranks 3 and 4; and finally group $G(\lfloor \log(q+1) \rfloor + 1)$ contains the largest possible rank, $\lfloor \log(q+1) \rfloor + 1$, according to P3. See the table below.

Group	Ranks in the group
0	1
1	2
2	3, 4
3	5, ..., 16
...	...
$G(\lfloor \log(q+1) \rfloor + 1)$..., $\lfloor \log(q+1) \rfloor + 1$
g	$F(g-1) + 1, \dots, F(g)$

What is the total cost of p finds? For $find(i)$, the cost of the operation is the depth of i , i.e., # of nodes from i to the root. We apportion this cost between the find operation itself and certain nodes along the path from i to the root using the following rules. Let j be any node on the path from i to the root.

R1: If j is the root, or $parent[j]$ is the root, or $parent[j]$ is in a different rank group from j , assign 1 unit of cost to the find operation.

R2: If j and $parent[j]$ are in the same rank group and $parent[j]$ is not the root, assign 1 unit of cost to j .

The figure below contains an example of apportioning cost.

f7.pdf

The total cost incurred by p finds is the total cost assigned to p finds by R1, denoted $cost(R1)$, plus the total cost assigned to all nodes by R2, denoted $cost(R2)$.

$$\begin{aligned}
 cost(R1) &\leq p \cdot (\text{max. cost assigned to any find}) \\
 &\leq p \cdot (\# \text{ of different rank groups} + 1) \\
 &\leq p \cdot (G(\lfloor \log(q+1) \rfloor + 1) + 1 + 1) \\
 &\leq O(pG(\log q)).
 \end{aligned}$$

$$\begin{aligned}
 cost(R2) &= \sum_{\forall g} \text{total cost assigned to nodes by R2 in group } g \\
 &\leq \sum_{\forall g} (\# \text{ of nodes in group } g) \cdot (\text{max. cost assigned to any node by R2 in group } g) \\
 &\leq \sum_{g=0}^{G(\lfloor \log(q+1) \rfloor + 1)} N(g) \cdot C_{max}(g) \\
 &\leq \sum_{g=0}^{G(\lfloor \log(q+1) \rfloor + 1)} \frac{2n}{F(g)} \cdot F(g) \\
 &= 2n(G(\lfloor \log(q+1) \rfloor + 1) + 1) \\
 &= O(nG(\log q)).
 \end{aligned}$$

What is $N(g)$, # of nodes in rank group g ?

$$\begin{aligned}
 N(g) &\leq \sum_{r=F(g-1)+1}^{F(g)} \frac{n}{2^{r-1}} \quad (\text{By P4}) \\
 &\leq \frac{n}{2^{F(g-1)}} \left(1 + \frac{1}{2} + \frac{1}{4} + \dots\right) \\
 &\leq \frac{2n}{F(g)}.
 \end{aligned}$$

What is $C_{max}(g)$, max. cost assigned to any node by R2 in rank group g ?

Assume for node i , its $r(i)$ is in group g . So $F(g-1) + 1 \leq r(i) \leq F(g)$. Recall that there are $F(g) - F(g-1)$ ranks in group g . Every time i is assigned 1 unit of cost by R2, i is reconnected to a new parent with a higher rank than its previous parent (by P5). Node i will be assigned and moved at most $F(g) - F(g-1)$ times until it is connected to a parent in a different rank group or to the root. So $C_{max}(g) \leq F(g) - F(g-1) \leq F(g)$.

The total cost for executing σ is the sum of (1) $O(q)$ for q unions; and (2) $O(pG(\log q)) + O(nG(\log q))$ for p finds. So the total is $O(q + pG(\log q) + nG(\log q))$, almost a linear function since $G(\cdot)$ is close to a constant function.

3 Greedy Algorithms

3.1 Introduction

Reading: CLRS 16.2

- Making change: How to pay x to a customer using the smallest possible number of coins, given the coinage, $c_1 < c_2 < \dots < c_n$, and a unlimited supply of coins of each denomination?

A greedy algorithm: Starting with nothing, at every stage we add to the coins already chosen a coin of the largest value available that does not take us past the amount to be paid.

Consider the American coinage. If $x = 94$, then the change includes 3 quarters, 1 dime, and 1 nickel and 4 pennies. A proof, though surprisingly hard, exists that the proposed greedy algorithm always gives the optimal (minimum) solution for the American coinage.

The algorithm may not be optimal for other coinages. For example, let $c_1 = 1$, $c_2 = 9$, and $c_3 = 15$. Assume $x = 19$. The greedy algorithm gives a solution with 5 coins. However, the optimal solution has only 3 coins.

- Greedy algorithms are suitable for solving optimization problems with solutions consisting of a sequence of choices. Greedy algorithms makes the choice that looks the best at the moment (short-sighted), and never changes the choices made (stubborn). So they may not always give optimal solutions.

Greedy(C : set) // C is the set of all choices

$S \leftarrow \emptyset$ // S is the solution set

while S is not a solution and $C \neq \emptyset$

$x \leftarrow$ an element in C that optimizes $Select(x)$ // You pick $Select(x)$ in advance

$C \leftarrow C - \{x\}$

if $S \cup \{x\}$ is a feasible solution then $S \leftarrow S \cup \{x\}$

if S is a solution then return S

else return "No solution found"

- Examples of greedy algorithms:

Kruskal's algorithm for finding the minimum spanning tree of an undirected edge-weighted graph.

Dijkstra's algorithm for finding the shortest paths from a source node to all other nodes in a directed, edge-weighted graph.

Heuristics for the knapsack problem.

Heuristics for the bin packing problem.

3.2 An activity-selection problem

Reading: CLRS 16.1

- Activity i has starting time s_i and finishing time f_i , with $s_i < f_i$. We say activities i and j are compatible if $f_i \leq s_j$ or $f_j \leq s_i$. Given $S = \{1, 2, \dots, n\}$, a set of proposed activities that wish to use a resource R which can only be used by one activity at a time. Determine $S' \subseteq S$ with the maximum size such that any two activities in S' are compatible.
- Greedy algorithm 1: Shortest activity first (SAF).
An instance that shows that SAF is not optimal.

i	s_i	f_i	$f_i - s_i$
1	0	3	3
2	3	6	3
3	2	4	2

- Greedy algorithm 2: Earliest starting activity first (ESAF).

An instance that shows that ESAF is not optimal.

i	s_i	f_i
1	1	3
2	0	5
3	4	6

- Greedy algorithm 3: Earliest finishing activity first (EFAF).

Theorem: EFAF is optimal.

Proof When multiple optimal solutions exist for some inputs, we may use "prove by transformation" to prove optimality of a greedy algorithm. Take any optimal schedule, assume that it is different from the greedy schedule. We wish to prove that the OPT schedule can be transformed to an EFAF schedule of the same number of activities. We use the following procedure to do the transformation.

```

 $S \leftarrow \{1, 2, \dots, n\}$ 
 $OPT \leftarrow \{a_1, a_2, \dots, a_l\}$ 
for  $i \leftarrow 1$  to  $l$ 
    if  $\exists j \in S - OPT$  such that  $f_j = \min_{k \in S - OPT} \{f_k\}$ ,  $f_j < f_{a_i}$ , and  $s_j \geq f_p$ ,
        where  $p$  is the immediate predecessor of  $a_i$  in the current  $OPT$ 
        then replace  $a_i$  with  $j$  in  $OPT$ , i.e.,  $OPT \leftarrow OPT - \{a_i\} \cup \{j\}$ 

```

By applying the above procedure to the OPT schedule, the resulting schedule is an EFAF schedule, with the same number of activities as in the OPT schedule. So EFAF is optimal.

Note: When the optimal solution for any input is unique, we may use "prove by contradiction" to prove optimality. Again, assume that the optimal solution is different from the greedy solution. Then show that by making some change to the optimal solution there is a better solution than the optimal. Thus, a contradiction.

3.3 Scheduling with deadlines

Reading: CLRS 16.5

- Given a single machine and n unit-time jobs, J_1, \dots, J_n . Assume that J_j has deadline d_j (integer) and penalty p_j (integer) which occurs when the deadline is missed. How can we schedule the jobs on the machine so that the total penalty is minimized?
- A greedy algorithm: Schedule jobs with the largest penalty in the latest possible time slot.

For the following instance, the greedy algorithm defines a schedule $J_4, J_2, J_3, J_1, J_7, J_6, J_5$ with total penalty of 50.

J_j	J_1	J_2	J_3	J_4	J_5	J_6	J_7
d_j	4	2	4	3	1	4	6
p_j	70	60	50	40	30	20	10

The following pseudocode describes the algorithm:

```

WLOG, assume  $p_1 \geq p_2 \geq \dots \geq p_n$ .
 $P \leftarrow 0$  //total penalty so far
for  $i \leftarrow 1$  to  $n$   $S[i] \leftarrow 0$  //S is the schedule
for  $j \leftarrow 1$  to  $n$ 
    if  $\exists k$  such that  $S[k] = 0$  and  $k \leq d_j$ 
        then  $i \leftarrow \max\{k | S[k] = 0 \text{ and } k \leq d_j\}$ 

```


else $i \leftarrow \max\{k | S[k] = 0\}$

$P \leftarrow P + p_j$

$S[i] \leftarrow j$

The worst-case time complexity of the greedy algorithm is $\Theta(n^2)$.

- Proof of optimality

Assume that the optimal schedule looks different from the greedy schedule. We wish to prove that the former can be transformed to the latter without changing the total penalty. We use c_j to denote the completion time of J_j in the current schedule, which is initially the optimal schedule. Assume $p_1 \geq p_2 \geq \dots \geq p_n$. For any J_j , it is either an early job or a late job in the current schedule. Consider the following procedure:

for $j \leftarrow 1$ to n

if J_j is an early job in the current schedule ($c_j \leq d_j$)

then $T \leftarrow \{J_i | J_i \in [c_j, d_j] \text{ and } i > j\}$

if $T \neq \emptyset$

then let J_i be the job in T with the largest c_i

swap J_i and J_j //step 1

else // J_j is a late job ($c_j > d_j$)

let J_i be the latest scheduled job with $i > j$

swap J_i and J_j //step 2

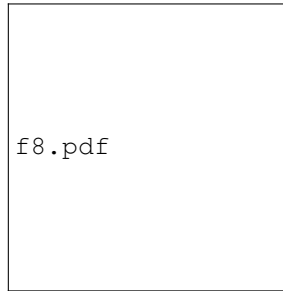
Step 1 does not increase the total penalty. This is because after the swap J_j remains to be an early job and J_i is moved to an earlier slot. Step 2 also does not increase the total penalty since J_j remains to be a late job and J_i is moved to an earlier slot. So after applying the above procedure, the optimal schedule is transformed to a greedy schedule while the total penalty is not increased. Since it is impossible to have a decreased total penalty, we have a greedy schedule with the same total penalty as the optimal schedule. Therefore, the greedy schedule is optimal.

4 Divide and Conquer

4.1 Introduction

Reading: CLRS 2.3

- A general template:



```
function D&C(x)
    if x is small and or simple then return adhoc(x)
    decompose x into smaller instances  $x_1, \dots, x_k$   $// k \geq 1$ 
    for  $i \leftarrow 1$  to  $k$   $y_i \leftarrow D\&C(x_i)$ 
    combine the  $y_i$ 's to obtain a solution y for x
    return y
```

Remarks:

- Relations between x and x_1, \dots, x_k , y_1, \dots, y_k and y ;
- Time complexity $T(n) = \sum_{i=1}^k T(n_i) + D(n) + C(n)$, where $D(n)$ and $C(n)$ are time for step 1 and step 3, respectively;
- Trade-off between $D(n)$ and $C(n)$;
- The time complexity requirement gives information on step 1 and step 3.

$O(\log n)$	$T(n) = T(\frac{n}{2}) + 1$
$O(n)$	$T(n) = 2T(\frac{n}{2}) + 1$ or $T(\frac{n}{2}) + n$
$O(n \log n)$	$T(n) = 2T(\frac{n}{2}) + n$ or $T(\frac{n}{2}) + n \log n$
$O(n^2)$	$T(n) = 4T(\frac{n}{2}) + n$ or $2T(\frac{n}{2}) + n^2$

- Examples of D&C algorithms:

Binary search: $T(n) = T(\frac{n}{2}) + O(1) \Rightarrow T(n) = O(\log n)$.

Merge sort: $T(n) = 2T(\frac{n}{2}) + O(n) \Rightarrow T(n) = O(n \log n)$, where $D(n) = O(1)$ and $C(n) = O(n)$.

Quick sort (best case): $T(n) = 2T(\frac{n}{2}) + O(n) \Rightarrow T(n) = O(n \log n)$, where $D(n) = O(n)$ and $C(n) = O(1)$.

Integer multiplication: $T(n) = 3T(\frac{n}{2}) + O(n) \Rightarrow T(n) = O(n^{\log 3}) = O(n^{1.585})$, where $D(n) = O(n)$ and $C(n) = O(n)$.

4.2 Matrix multiplication

Reading: CLRS 28.2

- Consider the multiplication of two $n \times n$ matrices. Using the definition of matrix multiplication, we need $\Theta(n^3)$. To use any divide-and-conquer idea, we have to first divide a matrix into several smaller matrices. Let $A_{n \times n}$ and $B_{n \times n}$ be the two matrices. Let $C_{n \times n} = A_{n \times n} \cdot B_{n \times n}$.

$$A_{n \times n} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B_{n \times n} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C_{n \times n} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then $C_{11} = A_{11}B_{11} + A_{12}B_{21}$, $C_{12} = A_{11}B_{12} + A_{12}B_{22}$, $C_{21} = A_{21}B_{11} + A_{22}B_{21}$, and $C_{22} = A_{21}B_{12} + A_{22}B_{22}$. So the multiplication of two $n \times n$ matrices becomes eight multiplications of two $\frac{n}{2} \times \frac{n}{2}$ matrices, giving us $T(n) = 8T(\frac{n}{2}) + \Theta(n^2)$. So $T(n) = \Theta(n^3)$ by master theorem. No improvement!

- In the late sixties, Strassen reduced eight multiplications to seven.

M_1	$(A_{12} - A_{22})(B_{21} + B_{22})$	C_{11}	$M_1 + M_2 - M_4 + M_6$
M_2	$(A_{11} + A_{22})(B_{11} + B_{22})$	C_{12}	$M_4 + M_5$
M_3	$(A_{11} - A_{21})(B_{11} + B_{12})$	C_{21}	$M_6 + M_7$
M_4	$(A_{11} + A_{12})B_{22}$	C_{22}	$M_2 - M_3 + M_5 - M_7$
M_5	$A_{11}(B_{12} - B_{22})$		
M_6	$A_{22}(B_{21} - B_{11})$		
M_7	$(A_{21} + A_{22})B_{11}$		

Using the above idea in the divide-and-conquer algorithm, we get $T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$, thus $T(n) = \Theta(n^{\log 7}) = \Theta(n^{2.81})$ by master theorem.

- In the late seventies, the matrix multiplication algorithm is improved to $\Theta(n^{2.61})$ by Pan. In the late eighties, the algorithm is improved to $\Theta(n^{2.376})$. There is still a substantial gap to the $\Omega(n^2)$ lower bound.

4.3 Finding the k th smallest

Reading: CLRS 8.2 and 8.3

- Given a list of n numbers, find the k th smallest number among them.
- First try: Sort the list in increasing order ($\Theta(n \log n)$) and locate the k th element ($\Theta(1)$).
- Second try: Similar to Quick Sort.

Function $Select(L, k)$

```

if  $|L| < 50$ 
    then sort  $L$  and return the  $k$ th
else choose any  $p \in L$  as a pivot
     $L_1 = \{a_i \in L | a_i < p\}$ 
     $L_2 = \{a_i \in L | a_i = p\}$ 
     $L_3 = \{a_i \in L | a_i > p\}$ 
    if  $k \leq |L_1|$  then return  $Select(L_1, k)$ 
    else if  $k \leq |L_1| + |L_2|$ 
        then return  $p$ 
    else return  $Select(L_3, k - |L_1| - |L_2|)$ 

```

Like Quick Sort, the time complexity of this algorithm heavily depends on the selection of the pivot in each recursion. If every time p happens to partition L evenly, then the time complexity is $\Theta(n)$. However, if the partition is extremely uneven, the time complexity degrades to $\Theta(n^2)$. Therefore, the worst-case time of the algorithm is $\Theta(n^2)$.

- Third try: Choose the pivot cleverly. Replace “choose any $p \in L$ as a pivot” in the above algorithm by the following code:

divide L into $\lfloor \frac{|L|}{5} \rfloor$ sublists of (up to) 5 elements each
 sort each sublist into increasing order
 let M be the list of medians of all sublists
 $p \leftarrow \text{Select}(M, \lceil \frac{|M|}{2} \rceil)$

From the illustration below, there will be at most $\frac{3}{4}|L|$ elements in L_1 and at most $\frac{3}{4}|L|$ elements in L_3 . Therefore, $T(n) \leq O(1)$ for $n \leq 49$ and $T(n) \leq T(\frac{n}{5}) + T(\frac{3n}{4}) + O(n)$ for $n \geq 50$. By induction, $T(n) = O(n)$. Since $\Omega(n)$ is obvious a lower bound, this D&C algorithm is optimal.

f9.pdf

- Theorem: Let $T(n) \leq cn$ for $n \leq 49$ and $T(n) \leq T(\frac{n}{5}) + T(\frac{3n}{4}) + cn$ for $n \geq 50$. Show that $T(n) \leq 20cn$.

Proof Induct on n . When $n \leq 49$, $T(n) \leq cn \leq 20cn$. Assume that $T(i) \leq 20ci$ for $i \leq n-1$. Now consider $T(n)$.

$$\begin{aligned}
 T(n) &\leq T(\frac{n}{5}) + T(\frac{3n}{4}) + cn \\
 &\leq 20c\frac{n}{5} + 20c\frac{3n}{4} + cn \\
 &= 4cn + 15cn + cn \\
 &= 20cn
 \end{aligned}$$

4.4 Exchanging two sections of an array

- Given an array A of n items. How can one exchange the first k items with the last $n-k$ items?
- A naive solution: copy the first k elements to an auxiliary array B ; move the last $n-k$ elements to the first $n-k$ positions of A ; and copy the k elements in B back to the last k positions of A .
- Assume only $\Theta(1)$ auxiliary memory is available. If the two sections have the same length, it is easy. For example, $A = (a, b, c, d, e, f)$ and $k = 3$. We can exchange the sections by using just one additional variable: swap a, d , swap b, e , and swap c, f .
- A D&C idea: Let $k = 3$. $(a, b, c, d, e, f, g, h, i, j, k) \rightarrow (d, e, f, g, h, i, j, k, a, b, c)$

f10.pdf

- Algorithm:

```

Swap(i, j, m) // Assume  $i + m \leq j$ 
    for  $p \leftarrow 0$  to  $m - 1$ 
        swap  $A[i + p]$  and  $A[j + p]$ 
Exchange(i, j, l, m) // Assume  $i + l \leq j$ 
    if  $l = m$  Swap(i, j, l)
    else if  $l < m$ 
        Swap(i, j + m - l, l)
        Exchange(i, j, l, m - l)
    else
        Swap(i, j, m)
        Exchange(i + m, j, l - m, m)

```

f11.pdf

- Time complexity: Let $T(l, m)$ be the number of single swaps. If $l = m$, $T(l, m) = l$; if $l < m$, $T(l, m) = l + T(l, m - l)$; and if $l > m$, $T(l, m) = m + T(l - m, m)$.

Prove by induction that $T(l, m) = l + m - \gcd(l, m)$.

Induct on $l + m$ (assuming $l, m > 0$). When $l + m = 2$, $l = m = 1$. So $T(l, m) = T(1, 1) = 1$ by definition. On the other hand, $l + m - \gcd(l, m) = 2 - \gcd(1, 1) = 1$. So the claim holds. In the inductive hypothesis, assume that for $l' + m' < l + m$, $T(l', m') = l' + m' - \gcd(l', m')$. Now consider the case of $l + m$. If $l = m$, $T(l, m) = l = 2l - l = l + m - \gcd(l, m)$. If $l < m$, $T(l, m) = l + T(l, m - l) = l + (l + m - l) - \gcd(l, m - l) = l + m - \gcd(l, m - l) = l + m - \gcd(l, m)$. If $l > m$, $T(l, m) = m + T(l - m, m) = m + (l - m + m) - \gcd(l - m, m) = l + m - \gcd(l, m)$.

- $Exchange(1, k + 1, k, n - k)$ solves the problem in time $T(k, n - k) = n - \gcd(k, n)$.

4.5 Computing exponentiation

- Idea:

$$a^n = \begin{cases} a & \text{if } n = 1 \\ (a^{\frac{n}{2}})^2 & \text{if } n \text{ is even} \\ a \times a^{n-1} & \text{otherwise} \end{cases}$$

For example, $a^{29} = aa^{28} = a(a^{14})^2 = a((a^7)^2)^2 = \dots = a((a(a(a^2)^2)^2)^2)^2$. It takes a total of seven multiplications.

For simplicity, we call this algorithm $expodac(a, n)$.

- Time complexity analysis:

Let $N(n)$ be the number of multiplications in $expodac(a, n)$.

$$N(n) = \begin{cases} 0 & \text{if } n = 1 \\ N(\frac{n}{2}) + 1 & \text{if } n \text{ is even} \\ N(n-1) + 1 & \text{otherwise} \end{cases}$$

When $n > 1$ is odd, $N(n) = N(n-1) + 1 = N(\frac{n-1}{2}) + 2 = N(\lfloor \frac{n}{2} \rfloor) + 2$. When n is even, $N(n) = N(\frac{n}{2}) + 1 = N(\lfloor \frac{n}{2} \rfloor) + 1$. Therefore, $N(\lfloor \frac{n}{2} \rfloor) + 1 \leq N(n) \leq N(\lfloor \frac{n}{2} \rfloor) + 2$. Define two functions N_i for $i = 1, 2$ as follows.

$$N_i(n) = \begin{cases} 0 & \text{if } n = 1 \\ N_i(\lfloor \frac{n}{2} \rfloor) + i & \text{otherwise} \end{cases}$$

It is easy to prove that $N_1(n) \leq N(n) \leq N_2(n)$. Since $N_1(n) = \Theta(\log n)$ and $N_2(n) = \Theta(\log n)$, then $N(n) = \Theta(\log n)$.

Now, if each integer multiplication can be done in constant time, the time complexity of $\text{expodac}(a, n)$ is $\Theta(\log n)$. But what if the integers involved are so large that a multiplication can not be completed in constant time? Let $M(p, q)$ be the time needed to multiply two integers of sizes p and q (the numbers of figures). Let $T(p, n)$ be the time complexity of $\text{expodac}(a, n)$, where p is the size of a .

Theorem/Exercise: The size of a^i is at least $i(p-1)$ but at most ip .

Inspection of $\text{expodac}(a, n)$ yields the following definition of $T(p, n)$.

$$T(p, n) \leq \begin{cases} 0 & \text{if } n = 1 \\ T(p, \frac{n}{2}) + M(\frac{pn}{2}, \frac{pn}{2}) & \text{if } n \text{ is even} \\ T(p, n-1) + M(p, p(n-1)) & \text{otherwise} \end{cases}$$

If $n > 1$ is odd, $T(p, n) \leq T(p, n-1) + M(p, p(n-1)) \leq T(p, \frac{n-1}{2}) + M(\frac{p(n-1)}{2}, \frac{p(n-1)}{2}) + M(p, p(n-1)) = T(p, \lfloor \frac{n}{2} \rfloor) + M(p, \lfloor \frac{n}{2} \rfloor) + M(p, p(n-1))$. If n is even, $T(p, n) = T(p, \frac{n}{2}) + M(\frac{pn}{2}, \frac{pn}{2}) \leq T(p, \lfloor \frac{n}{2} \rfloor) + M(p, \lfloor \frac{n}{2} \rfloor) + M(p, p(n-1))$. So in both cases,

$$T(p, n) \leq T(p, \lfloor \frac{n}{2} \rfloor) + M(p, \lfloor \frac{n}{2} \rfloor) + M(p, p(n-1)).$$

In general, $M(p, q) = \Theta(qp^{\alpha-1})$ where $p \leq q$ and $\alpha = 2$ in the classic integer multiplication algorithm and $\alpha = \log 3$ in the divide-and-conquer algorithm. So $M(p, \lfloor \frac{n}{2} \rfloor) = \Theta((p \lfloor \frac{n}{2} \rfloor)^\alpha)$ and $M(p, p(n-1)) = \Theta(p^\alpha(n-1))$. So,

$$T(p, n) \leq T(p, \lfloor \frac{n}{2} \rfloor) + \Theta(p^\alpha n^\alpha).$$

By the iteration method or the master method, we have $T(p, n) \leq \Theta(p^\alpha n^\alpha)$, thus $T(p, n) = O(p^\alpha n^\alpha)$.

On the other hand, consider the last or the next to last multiplication in $\text{expodac}(a, n)$, depending on whether n is even or odd. It involves squaring $a^{\lfloor \frac{n}{2} \rfloor}$, which is of size at least $(p-1)\lfloor \frac{n}{2} \rfloor$. So,

$$T(p, n) \geq M((p-1)\lfloor \frac{n}{2} \rfloor, (p-1)\lfloor \frac{n}{2} \rfloor).$$

Since $M((p-1)\lfloor \frac{n}{2} \rfloor, (p-1)\lfloor \frac{n}{2} \rfloor) = \Theta(((p-1)\lfloor \frac{n}{2} \rfloor)^\alpha)$, so $T(p, n) \geq \Theta(p^\alpha n^\alpha)$, thus $T(p, n) = \Omega(p^\alpha n^\alpha)$.

Combining the two bounds, we have $T(p, n) = \Theta(p^\alpha n^\alpha)$, where $\alpha = 2$ if the classic algorithm is used and $\alpha = \log 3$ if the divide-and-conquer algorithm is used.

4.6 The closest-pair problem

Reading: CLRS 33.4

- Problem:

Given n 2D points, find the two closest points.

Remarks:

- Input is $(x_1, y_1), \dots, (x_n, y_n)$ and output is $(x_i, y_i), (x_j, y_j)$.
- The distance between $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is $|p_1 p_2| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.
- A brute-force algorithm: $\binom{n}{2} = O(n^2)$.
- A divide-and-conquer algorithm:

Let X be point set P sorted by increasing x and Y be point set P sorted by increasing y .

 1. Divide: Use a vertical line l to bisect P into P_L and P_R . X and Y are thus correspondingly partitioned into X_L and X_R , Y_L and Y_R .
 2. Conquer: Use two recursive calls to find the closest pairs in P_L and P_R . Let δ_L (δ_R) be the distance between the two closest points in P_L (P_R). Define $\delta = \min\{\delta_L, \delta_R\}$.
 3. Merge: Find the closest pair in P . It can be the pair with distance δ found in the previous step, or a pair with one point in P_L and the other in P_R with a distance less than δ .
- Time complexity: $T(n) = T_P(n) + T_{DAC}(n)$.
 - Preprocessing: Sort P twice to construct X and Y . $\Rightarrow O(n \log n)$
 - Divide: Use the median in X to create the partition of P into P_L and P_R . Construct the partition of X into X_L and X_R . (This is easy.) Construct the partition of Y into Y_L and Y_R . (This can be tricky.) All of the above must be done in linear time. To check whether you have the right partitions, are P_L , X_L , and Y_L the same point set, and are P_R , X_R , and Y_R the same point set? $\Rightarrow O(n)$
 - Conquer: Two recursive calls on point sets of size $\frac{n}{2}$. $\Rightarrow 2T_{DAC}(\frac{n}{2})$
 - Merge: Many technical details to fill in. We wish to spend only linear time for the merge. Can we achieve this goal? $\Rightarrow O(n)$

So $T_{DAC}(n) = 2T_{DAC}(\frac{n}{2}) + O(n) = O(n \log n)$. Overall, $T(n) = T_P(n) + T_{DAC}(n) = O(n \log n) + O(n \log n) = O(n \log n)$.

- Merge in linear time:

Assume that

- $P \Rightarrow P_L, P_R$, $X \Rightarrow X_L, X_R$, $Y \Rightarrow Y_L, Y_R$ by the vertical line l .
- δ_L is the distance between the closest points in P_L and δ_R is the distance between the closest points in P_R .
- $\delta = \min\{\delta_L, \delta_R\}$.

Goal: Determine if there are two points, one in P_L and the other in P_R , with distance less than δ .

An exhaustive search checks all pairs and may take $O(n^2)$. Can we just check $O(n)$ pairs and not miss any one with distance less than δ ? Yes and here is why.

Define a strip centered at l with width 2δ . Let P_S be the set of points in the strip and Y_S be P_S sorted by y . (Remember our linear time restriction: Can you create P_S and Y_S in linear time?)

Claim: If there are points $p \in P_L$ and $q \in P_R$ with $|pq| < \delta$, then p and q must be in the strip.

Pause: Can we check out all pairs in P_S to determine the one with the smallest distance?

For each $p \in Y_S$, define a rectangle $R(p)$ of height δ and width 2δ , with the bottom edge of the rectangle passing p .

Claim: If there are p and q with $|pq| < \delta$ and $q.y \geq p.y$, q must be in $R(p)$.

Claim: There can be at most eight points in each $R(p)$.

Why? Divide $R(p)$ ($\delta \times 2\delta$) into eight $\frac{\delta}{2} \times \frac{\delta}{2}$ squares. In each square, if there are two or more points, say q_1 and q_2 , then

$$|q_1 q_2| \leq \text{diagonal of the square} = \sqrt{2} \frac{\delta}{2} < \delta,$$

which is impossible since q_1 and q_2 are on the same side of the vertical line l . So there can be at most one point in each square, with a total of eight points in $R(p)$.

Claim: For any $p \in Y_S$, if there is q such that $|pq| < \delta$, then q must be one of the seven points following p in Y_S .

Algorithm for merge:

```

m = |Ys|
mindist = |Ys[0]Ys[1]|
p = Ys[0]
q = Ys[1]
for i = 1 to m-1
    k = min {i + 7, m}
    for j = i + 1 to k
        dist = |Ys[i]Ys[j]|
        if dist < mindist
            mindist = dist
            p = Ys[i]
            q = Ys[j]
If mindist < delta
    return p and q as the closest points
else return the closest points found by
    the recursive calls

```


5 Dynamic Programming

5.1 Introduction

Reading: CLRS 15.3

- Divide-and-conquer algorithms are implemented by recursion. Its design is top-down, and it is efficient when the subproblems don't overlap. However, when subproblems do overlap (share sub-subproblems), recursion does redundant work. In this case, a tabular method is often used. It is nonrecursive and bottom-up. It is called dynamic programming.
- An example: Fibonacci numbers:

```
fib1(n)
  if n < 2 return n
  else return fib1(n-1) + fib1(n-2)
```

We can see that this recursive (divide-and-conquer) algorithm is not efficient. To compute $\text{fib1}(n)$, the algorithm computes $\text{fib1}(n-1)$ and $\text{fib1}(n-2)$ separately. To compute $\text{fib1}(n-1)$, the values of $\text{fib1}(n-2)$ and $\text{fib1}(n-3)$ are needed. To compute $\text{fib1}(n-2)$, the values of $\text{fib1}(n-3)$ and $\text{fib1}(n-4)$ are needed. We observe that subproblems $\text{fib1}(n-1)$ and $\text{fib1}(n-2)$ share sub-subproblem.

The time complexity $T(n) \geq T(n-1) + T(n-2)$. So $T(n)$ is larger than the n th Fibonacci number. So $T(n) \geq \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (-\frac{1+\sqrt{5}}{2})^{-n}) = \Theta(1.618^n)$.

We can use the dynamic programming method by building a 1-D table as below and returning the n th entry of the table.

k	0	1	2	3	4	...	n
f_k	0	1	1	2	3	...	f_n

```
fib2(n)
  if n < 2 return n
  else i ← 0
      j ← 1
      for k ← 2 to n
        f ← i + j
        i ← j
        j ← f
      return f
```

The time complexity is obviously $O(n)$.

To summarize, to use dynamic programming, first define a function F recursively (so that the solution information is embedded in $F(n)$): $F(n) = G(F(n_1), F(n_2), \dots, F(n_k))$ for $n_1, n_2, \dots, n_k < n$. Construct a table to compute nonrecursively $F(n_1), F(n_2), \dots, F(n_k)$, hence $F(n)$.

5.2 Traveling salesman problem (TSP)

- Given a edge-weighted graph $G = (V, E)$. Find a tour (a cycle that passes through each vertex exactly once) with the minimum total weight.
- Assume $V = \{1, 2, \dots, n\}$. Let $S \subseteq \{2, 3, \dots, n\}$ and $i \notin S$. Define $C(S, i)$ = minimum total weight of simple paths from 1 to all nodes in S and to i . First, $C(\emptyset, i) = w(1, i)$, where $w(1, i) = \infty$ if $(1, i) \notin E$. Second, $C(S, i) = \min_{k \in S} \{C(S - \{k\}, k) + w(k, i)\}$ if $S \neq \emptyset$. Then the total weight of the traveling salesman tour is $C(\{2, 3, \dots, n\}, 1)$.

- If using recursion to compute C ,

$$\begin{aligned}
T(n) &\geq (n-1)(T(n-1) + 1) \\
&= (n-1)T(n-1) + (n-1) = (n-1)(n-2)T(n-2) + (n-1)(n-2) + (n-1) \\
&= \dots \\
&= (n-1)!T(1) + (n-1)! + \frac{(n-1)!}{1!} + \frac{(n-1)!}{2!} + \dots + \frac{(n-1)!}{(n-2)!} \\
&= (n-1)!\left(1 + 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots + \frac{1}{(n-2)!}\right) \\
&> 3(n-1)! \\
&= \Omega((n-1)!).
\end{aligned}$$

- If using dynamic programming,

$S \setminus i$	1	2	...	n
\emptyset	\rightarrow	\rightarrow	...	\rightarrow
...
$\{2, 3, \dots, n\}$	*

- The arrows indicate the computing order: row by row and left to right. The number of rows is the same as the number of subsets of $\{2, 3, \dots, n\}$, which is $\binom{n-1}{0} + \binom{n-1}{1} + \dots + \binom{n-1}{n-1} = 2^{n-1}$. Also there are n columns. To compute each entry $C[S, i]$, we need $O(|S|) = O(n)$ steps. So the total time needed is $T(n) = O(n^2 2^{n-1})$. An improvement over the recursive implementation.

5.3 Chained matrix multiplication

Reading: CLRS 15.2

- We wish to compute $A_1 \times A_2 \times \dots \times A_n$, where A_i is a $p_{i-1} \times p_i$ matrix. Which order of computation should we use to achieve the highest efficiency of the algorithm?
- The number of basic operations needed to compute $A_i \times A_{i+1}$ is $p_{i-1}p_i p_{i+1}$.
- Order of computation determines the time efficiency. For example, $A_1 : 10 \times 20$, $A_2 : 20 \times 50$, $A_3 : 50 \times 1$, and $A_4 : 1 \times 100$. If we use the order in $A_1 \times (A_2 \times (A_3 \times A_4))$, the number of basic operations is $(50 \times 1 \times 100) + (20 \times 50 \times 100) + (10 \times 20 \times 100) = 125,000$. However, if we use the order in $((A_1 \times A_2) \times A_3) \times A_4$, the number of basic operations is $(10 \times 20 \times 50) + (10 \times 50 \times 1) + (10 \times 1 \times 100) = 11,500$.
- Question: What is the minimum number of basic operations in computing $A_1 \times A_2 \times \dots \times A_n$?
- Let $m(i, j)$ be the minimum number of basic operations in computing $A_i \times A_{i+1} \times \dots \times A_j$ for $1 \leq i \leq j \leq n$. Assume in general that k is used to indicate the position of the last multiplication to be performed among all: $(A_i \times \dots \times A_k) \times (A_{k+1} \times \dots \times A_j)$. Then

$$m(i, j) = 0 \text{ if } i = j.$$

$$m(i, j) = \min_{i \leq k \leq j-1} \{m(i, k) + m(k+1, j) + p_{i-1}p_k p_j\} \text{ if } i \neq j.$$
- We can use a dynamic programming algorithm to compute $m(1, n)$, the minimum number of basic operations in computing $A_1 \times A_2 \times \dots \times A_n$. Entries are filled left to right and bottom to top. Note that those in the lower left triangle are undefined.

$i \setminus j$	1	2	...	n
1	0	\uparrow	...	*
2	—	0	...	\uparrow
...
n	—	—	...	0

- The algorithm:

```

for  $i \leftarrow 1$  to  $n$ 
     $m[i, i] \leftarrow 0$ 
for  $j \leftarrow 2$  to  $n$ 
    for  $i \leftarrow j - 1$  to  $1$ 
         $m[i, j] \leftarrow \min_{i \leq k \leq j-1} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\}$ 

```

- Time complexity: $O(n^3)$.

5.4 Making change

- Let n , a positive integer, be the number of different types of coin in a country. Let $coin[1..n]$, an array of positive integers, be the values of these n types of coin. Let m , a positive integer, be the amount of change that one wishes to make. Design a dynamic programming algorithm that determines whether m can be made with the coins, and if so, computes the minimum number of coins needed.

- Define $count(i)$ to be the minimum number of coins to make i (> 0). That $count(i) = \infty$ implies that no solution exists. The recursive definition of $count(i)$ is as follows.

$count(1) = \infty$ if $1 \notin coin[]$.

$count(coin[j]) = 1$ for $j = 1, \dots, n$.

$count(i) = 1 + \min_{1 \leq j \leq n, coin[j] < i} \{count(i - coin[j])\}$

- The table is a 1-D table and its entries are filled from left to right until $count[m]$ is reached.

i	1	2	\dots	m
$count[i]$	\rightarrow	\rightarrow	\dots	*

- Algorithm:

```

for  $i \leftarrow 1$  to  $m$   $count[i] \leftarrow -1$ 
 $count[1] \leftarrow \infty$ 
for  $j \leftarrow 1$  to  $n$ 
     $count[coin[j]] \leftarrow 1$ 
for  $i \leftarrow 1$  to  $m$ 
    if  $count[i] = -1$ 
         $min \leftarrow \infty$ 
        for  $j \leftarrow 1$  to  $n$ 
            if  $coin[j] < i$ 
                if  $min > count[i - coin[j]]$ 
                     $min \leftarrow count[i - coin[j]]$ 
         $count[i] \leftarrow 1 + min$ 

```

- Time complexity: $\Theta(mn)$. (pseudo-polynomial)

5.5 Longest common subsequence

Reading: CLRS 15.4

- Subsequence: If $X = \langle A, B, C, B, D, A, B \rangle$ and $Z = \langle B, C, D, B \rangle$, then Z is a subsequence of X .

Common subsequence: Let $Y = \langle B, D, C, A, B, A \rangle$. Then $\langle B, C, A \rangle$ is a common subsequence of X and Y .

Longest common subsequence (LCS): For X and Y , there is no common subsequence with length longer than 4. $\langle B, C, B, A \rangle$ and $\langle B, D, A, B \rangle$ are both LCS's of X and Y .

Question: Given two sequences, what is the length of their LCS? (What is the LCS of the sequences?)

- A brute-force method:

Assume $X = \langle x_1, \dots, x_m \rangle$ and $Y = \langle y_1, \dots, y_n \rangle$. For each subsequence of X , check if it is also a subsequence of Y , keeping track of the longest found.

How many possible subsequences are there for X ? $\binom{m}{0} + \binom{m}{1} + \binom{m}{2} + \dots + \binom{m}{m} = 2^m$.

- A recursive approach:

Define $X_i = \langle x_1, \dots, x_i \rangle$ and $Y_j = \langle y_1, \dots, y_j \rangle$. Define $C(i, j)$ to be the length of the LCS of X_i and Y_j .

$C(i, j) = 0$ if $i = 0$ or $j = 0$;

$C(i, j) = C(i - 1, j - 1) + 1$ if $i, j > 0$ and $x_i = y_j$;

$C(i, j) = \max\{C(i, j - 1), C(i - 1, j)\}$ if $i, j > 0$ and $x_i \neq y_j$.

When $x_i = y_j$, $X_i = X_{i-1} \langle x_i \rangle$ and $Y_j = Y_{j-1} \langle y_j \rangle$. So $LCS(X_i, Y_j) = LCS(X_{i-1}, Y_{j-1}) \langle x_i \rangle$. Hence, $C(i, j) = C(i - 1, j - 1) + 1$.

When $x_i \neq y_j$, x_i and y_j cannot both appear in $LCS(X_i, Y_j)$. So $LCS(X_i, Y_j) = LCS(X_i, Y_{j-1})$ or $LCS(X_{i-1}, Y_j)$. Hence, $C[i, j] = \max\{C(i, j - 1), C(i - 1, j)\}$.

- A nonrecursive implementation: Dynamic programming:

A 2-D table is constructed where each entry is filled left to right and top to bottom. The initialization handles the first row and the first column of the table. Entry $C[m, n]$ is the length of the LCS of X and Y .

$i \backslash j$	0	1	2	...	n
0	0	0	0	...	0
1	0	\rightarrow	\rightarrow	...	\rightarrow
2	0	\rightarrow	\rightarrow	...	\rightarrow
...
m	0	\rightarrow	\rightarrow	...	*

- The algorithm:

for $i \leftarrow 0$ to m $C[i, 0] \leftarrow 0$

for $j \leftarrow 0$ to n $C[0, j] \leftarrow 0$

for $i \leftarrow 1$ to m

for $j \leftarrow 1$ to n

if $x_i = y_j$ $C[i, j] \leftarrow C[i - 1, j - 1] + 1$

else $C[i, j] = \max\{C[i, j - 1], C[i - 1, j]\}$

- Time complexity: $\Theta(mn)$.

- How to compute the LCS in addition to the length of the LCS: Maintain an array $S[i, j]$ of special characters. Set $S[i, 0] = S[0, j] = \sqcup$ (single space) for $0 \leq i \leq m$ and $0 \leq j \leq n$. In the nested for loop, if $x_i = y_j$, set $S[i, j]$ to be \nwarrow , else if $C[i, j - 1] \geq C[i - 1, j]$, set $S[i, j]$ to be \leftarrow , and if $C[i, j - 1] < C[i - 1, j]$, set $S[i, j]$ to be \uparrow . The following additional code generates the LCS of two sequences.

```

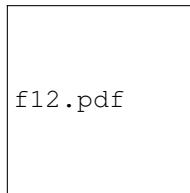
 $i \leftarrow m$ 
 $j \leftarrow n$ 
while  $S[i, j] \neq \sqcup$ 
    if  $S[i, j] = \leftarrow j \leftarrow j - 1$ 
    else if  $S[i, j] = \uparrow i \leftarrow i - 1$ 
    else push  $x_i$  to a stack
         $i \leftarrow i - 1$ 
         $j \leftarrow j - 1$ 
output the content in the stack

```

5.6 Optimal binary search tree

Reading: CLRS 15.5

- Given a set of keys (numbers) and the probability that each key is located. How can one organize the set in a binary search tree so that the average time to locate a key in the tree is minimized?
- For each node (key) in a binary search tree, the time needed to locate the node is its level number.
- Let the keys be a_1, a_2, \dots, a_n (in increasing order). Let l_i be the level number of the node corresponding to key a_i in a given binary search tree. Let p_i be the probability that a_i is to be located. Then the average search time for that tree is $\sum_{i=1}^n p_i l_i$. We wish to build an optimal binary search tree, where this cost is minimized.
- An example: $n = 3$ and $p_1 = 0.7$, $p_2 = 0.2$ and $p_3 = 0.1$. The following figure contains all five possible binary search trees for $n = 3$.



- $3(0.7) + 2(0.2) + 1(0.1) = 2.6$
- $2(0.7) + 3(0.2) + 1(0.1) = 2.1$
- $2(0.7) + 1(0.2) + 2(0.1) = 1.8$
- $1(0.7) + 3(0.2) + 2(0.1) = 1.5$
- $1(0.7) + 2(0.2) + 3(0.1) = 1.4 \leftarrow \text{optimal!}$

- A recursive approach: Let $c(i, j)$ be the average search time in a tree with only a_i, \dots, a_j , where $1 \leq i \leq j \leq n$. If a_k happens to be the root of the tree containing a_i, \dots, a_j , then in the left subtree are a_i, \dots, a_{k-1} and in the right subtree are a_{k+1}, \dots, a_j .

$$c(i, i) = p_i \text{ for } 1 \leq i \leq n$$

$$c(i, j) = \min_{i \leq k \leq j} \{c(i, k-1) + c(k+1, j) + \sum_{l=i}^j p_l\} \text{ for } i < j$$

$$c(i, j) = 0 \text{ for } i = j + 1 \text{ (Why needed?)}$$

- A dynamic programming algorithm with $O(n^3)$:

$i \backslash j$	1	2	...	$n-1$	n
1	p_1	\uparrow	...	\uparrow	*
2	0	p_2	...	\uparrow	\uparrow
...
n	---	---	...	0	p_n

5.7 Memory functions

Reading: CLRS 15.3

- Divide and conquer: Only needed entries are computed but some entries are computed more than once. Dynamic programming: All entries in the table are computed once, whether needed or not.
- A compromise: Only compute needed entries exactly once. To do so, we combine the recursive implementation with a table. Before we enter a recursion, we check the table to see whether the entry has been computed before. This method is called the memory function method.
- Example: Chained matrix multiplication revisited.

We first initialize all entries in table $m[1..n, 1..n]$ to be -1 , and then call $mf(1, n)$.

```
mf(i, j)
    if  $i = j$  return 0
    if  $m[i, j] \neq -1$  return  $m[i, j]$ 
     $c \leftarrow \infty$ 
    for  $k \leftarrow i$  to  $j - 1$ 
         $c \leftarrow \min\{c, mf(i, k) + mf(k + 1, j) + p_{i-1}p_kp_j\}$ 
     $m[i, j] \leftarrow c$ 
    return  $c$ 
```

- The time complexity is no larger than that in the corresponding dynamic programming algorithm, but the space complexity will be more since recursion requires more space to implement.

6 The Lower Bound Theory

6.1 What is a lower bound?

- Algorithmics/upper bounds:

Prove that a problem can be solved in time $O(f(n))$ by designing and analyzing a specific algorithm for the problem, for some $f(n)$ that we aim to reduce as much as possible. We say that $O(f(n))$ is an upper bound to the problem. An upper bound is always paired with an algorithm. It is the amount of time sufficient to solve the problem.

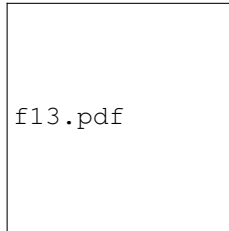
- Complexity/lower bounds:

Prove that any algorithm capable of solving the problem correctly on all of its instances must take time $\Omega(g(n))$, for some $g(n)$ that we try to push as large as possible. We say that $g(n)$ is a lower bound to the problem. A lower bound shows the difficult nature of the problem, so it has nothing to do with any specific algorithm. It is the amount of time necessary to solve the problem by any algorithm within the model.

- Optimality of an algorithm:

If $f(n) = \Theta(g(n))$, then we say that we have found the most efficient algorithm possible, except perhaps for changes in the hidden multiplicative constant.

- A graphical explanation:



- Example: Matrix multiplication.

Lower bound: n^2

Upper bounds: $n^3 \rightarrow n^{2.81} \rightarrow n^{2.61} \rightarrow n^{2.376} \rightarrow \dots$

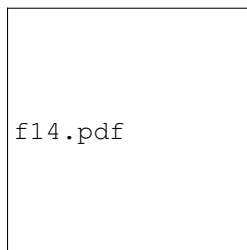
6.2 The decision tree method

Reading: CLRS 8.1

- A comparison-based model for proving worst-case lower bound:

The following is a decision tree for sorting 3 items by comparisons. Each interior node represents a comparison. Each tree edge is an outcome ($<$ or $>$). Each leaf is a possible output (a sorted list). Each path from the root to a leaf represents the steps of sorting a certain type of input. There are $n!$ leaves in the tree, corresponding to $n!$ permutations. The height of the tree gives the maximum number of comparisons in the algorithm for all inputs, thus the worst case time complexity of the algorithm the tree represents.

How to use a decision tree to establish a lower bound: For each comparison-based sorting algorithm, A_i , there is a decision tree, T_i . Let $height(T_i)$ be the height of T_i , thus the worst-case time of A_i . Obviously, $\min_i \{height(T_i)\}$ is a lower bound to the problem.



- Lemma: A binary tree of height h has at most 2^h leaves.

Proof We assume that the height of a tree is the number of edges on the longest path between the root and a leaf. We induct on h . When $h = 0$, the tree has just one node. There is 1 leaf, which is no larger than 2^0 . Assume for $i \leq h - 1$, a tree of height i has no more than 2^i leaves. Now consider a tree with height h . Assume the heights of the left and right subtrees are h_L and h_R , respectively. so $h = \max\{h_L, h_R\} + 1$. By the inductive hypothesis, the numbers of leaves in the left and right subtrees are at most 2^{h_L} and 2^{h_R} , respectively. So the number of leaves in the tree with height h is at most $2^{h_L} + 2^{h_R} \leq 2^h$.

- Theorem: Any comparison-based sorting algorithm requires $\Omega(n \log n)$ time in the worst case.

Proof Let T be a decision tree for any algorithm that sorts n elements by comparisons. T has $n!$ leaves. So the height of the tree, h , is at least $\log n!$. (Assume not. Then $h < \log n!$. By the lemma, the number of leaves in the tree is at most $2^h < 2^{\log n!} = n!$, which is impossible.) So for any sorting algorithm, the worst-case time complexity is the height of the decision tree, thus at least $\log n! = \Omega(n \log n)$.

- Both Merge Sort and Heap Sort achieve the optimality of sorting since their worst-case times match the established lower bound, $n \log n$. Radix Sort has a worst-case time of $\Theta(d(n+k))$, where d is the number of digits and k is the base. It is $\Theta(n)$ when $d = O(1)$ and $k = O(n)$. This is not a contradiction to the $n \log n$ lower bound, since Radix Sort is not comparison based.

- A comparison-based model for proving average-case lower bound:

Again consider the example of sorting 3 elements. As in any average-case analysis, assume that all elements in the list are distinct and that all permutations are equally likely. For any sorting algorithm, there is a corresponding decision tree with $n!$ leaves. Let d_i be the depth, or the number of edges on the path between the root and a leaf i . Let p_i be the probability that permutation (leaf) i occurs. Then the average-case time of the algorithm that the tree represents is $\sum_i p_i d_i = \frac{1}{n!} \sum_i d_i$.

leaf	abc	acb	cab	bac	bca	cba
probability	p_1	p_2	p_3	p_4	p_5	p_6
depth	d_1	d_2	d_3	d_4	d_5	d_6

- Lemma: Let T_m be any binary tree with m leaves such that each nonleaf has exactly 2 children. Define $D(T_m) = \sum_i d_i$ (the sum of the depths of all leaves in T). Define $D_m = \min_{T_m} \{D(T_m)\}$. Prove that $D_m \geq m \log m$.

Proof We induct on m . When $m = 1$, the tree has only one node. So $D_1 = 0 \geq 1 \log 1$. Assume that for any $i \leq m - 1$, $D_i \geq i \log i$. Now consider any tree with m leaves. Assume there are i leaves in the left subtree and $m - i$ leaves in the right subtree. ($i \neq 0$ and $i \neq m$ since each nonleaf has two children.) So

$$\begin{aligned} D_m &= \min_{1 \leq i \leq m-1} \{D_i + D_{m-i} + m\} \\ &\geq m + \min_{1 \leq i \leq m-1} \{i \log i + (m-i) \log(m-i)\} \end{aligned}$$

Define function $f(x) = x \log x + (m-x) \log(m-x)$. Then the derivative $f'(x) = x \frac{1}{\ln 2} \frac{1}{x} + \log x + (m-x) \frac{1}{\ln 2} \frac{-1}{m-x} - \log(m-x) = \log x - \log(m-x)$. Let $f'(x) = 0$. Then $\log x = \log(m-x)$, which implies $x = \frac{m}{2}$. So $f(x)$ achieves the minimum when $x = \frac{m}{2}$. To continue,

$$\begin{aligned} D_m &\geq m + \min_{1 \leq i \leq m-1} \{i \log i + (m-i) \log(m-i)\} \\ &\geq m + \frac{m}{2} \log \frac{m}{2} + \frac{m}{2} \log \frac{m}{2} \\ &= m + m \log \frac{m}{2} \\ &= m \log m \end{aligned}$$

- Theorem: Any comparison-based sorting algorithm requires $\Omega(n \log n)$ time in the average case.

Proof Let T be a decision tree for any algorithm that sorts n elements by comparisons. T has $n!$ leaves. So the average depth of the leaves is $\frac{1}{n!} \sum_i d_i \geq \frac{1}{n!} D_m \geq \frac{1}{n!} n! \log n! = \log n!$. So for any sorting algorithm, the average-case time complexity is the average depth of the decision tree, thus at least $\log n! = \Omega(n \log n)$.

- Bucket Sort: not comparison-based, with $O(n)$ average time.

Let $A = (a_1, \dots, a_n)$, where $a_i \in [0, 1)$ is generated randomly such that a_1, \dots, a_n are distributed uniformly in $[0, 1)$. For example, $A = (0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.61)$. Let B be an array of intervals in $[0, 1)$, where the length of each interval is $\frac{1}{n}$. In particular, $B[i]$ represents interval $[\frac{i}{n}, \frac{i+1}{n})$ for $0 \leq i \leq n-1$. Each $B[i]$ will be a pointer to a list of numbers falling into the corresponding interval.

for $i \leftarrow 1$ to n

 insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$

for $i \leftarrow 0$ to $n-1$

 sort list $B[i]$ using any sorting algorithm

concatenate $B[0], \dots, B[n-1]$ into a list for output

Why should $A[i]$ go to list $B[\lfloor nA[i] \rfloor]$? Assume $A[i]$ goes to $B[x]$. Then $\frac{x}{n} \leq A[i] < \frac{x+1}{n}$. So $x \leq nA[i] < x+1$. So $nA[i] - 1 < x \leq nA[i]$. Since x is an integer, $x = \lfloor nA[i] \rfloor$.

Time complexity: $\Theta(n^2)$ or $\Theta(n \log n)$ in the worst case; $\Theta(n)$ in the average case since each list in B has 1 element on average.

6.3 The adversary method

- The adversary strategy: An adversary provides worst-case input that forces all algorithms solving the problem to work as hard as possible.

Example: Alice (1) has a date (month/day) in mind (this is what she tells Bob); (2) answers Bob's questions honestly; and (3) wants to force Bob to ask as many questions as possible. Bob (1) asks yes/no questions and (2) wants to guess the date as quickly as possible.

Bob: Is it in the winter? (December, January and February)

Alice: No (since there are more dates in the other three seasons).

Bob: Is the first letter of the month's name in the first half of the alphabet?

Alice: Yes (since among the remaining 9 months, 6 are in the first half of the alphabet, March, April, May, June, July, August).

.....

Alice didn't pick a date at all. Is this cheating? Not to Bob. In fact, Alice will not do so until the need for consistency in her answers pins her down. This is how a good adversary should be.

For a comparison-based model, the adversary works as follows:

f15.pdf

- Finding the max and min at the same time:

Let $C_{1,n}(n)$ be the number of comparisons necessary and sufficient to find the maximum and minimum of n integers. Prove that $C_{1,n}(n) = \lceil \frac{3n}{2} \rceil - 2$.

Upper bound: $C_{1,n}(n) \leq \lceil \frac{3n}{2} \rceil - 2$. Wish to design an algorithm which takes at most $\lceil \frac{3n}{2} \rceil - 2$ comparisons. The algorithm first uses $n-1$ comparisons to find the maximum among the n numbers and then $\lceil \frac{n}{2} \rceil - 1$ more comparisons to find the minimum among the $\lceil \frac{n}{2} \rceil$ non-winners in the first round.

Lower bound: $C_{1,n}(n) \geq \lceil \frac{3n}{2} \rceil - 2$. Assume all keys are distinct. At any time in any algorithm, there are four kinds of elements.

Novice: not compared yet;

Winner: always wins so far;

Loser: always loses so far;

Moderate: wins some and loses some.

Define a state (i, j, k, l) , where i, j, k , and l are the numbers of novices, winners, losers, and moderates, respectively. The initial state is $(n, 0, 0, 0)$ and the final state is $(0, 1, 1, n-2)$. Question: What is the minimum number of comparisons from the initial state to the final state? A state transition table is given below:

	(i, j, k, l)
NN	$(i-2, j+1, k+1, l)$
NW	$(i-1, j, k, l+1)$ or $(i-1, j, k+1, l)$
NL	$(i-1, j+1, k, l)$ or $(i-1, j, k, l+1)$
NM	---
WW	$(i, j-1, k, l+1)$
WL	(i, j, k, l) or $(i, j-1, k-1, l+2)$
WM	---
LL	$(i, j, k-1, l+1)$
LM	---
MM	---

Assume that an adversary provides the input and uses the following strategy:

algorithm	NN	NW	NL	WW	WL	LL
adversary	arbitrary	W wins	N wins	arbitrary	W wins	arbitrary

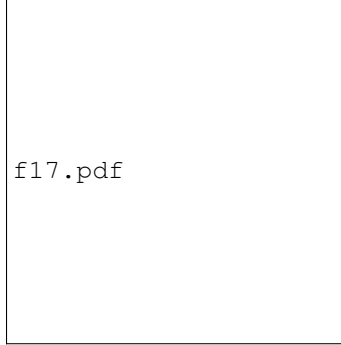
Why is the adversary able to make the choices it does in the table above? For an NW comparison, it can make N to be arbitrarily small so that W is the winner. For the similar reason in an NL comparison, it can make N to be arbitrarily large so that L is the loser. In a WL comparison, the adversary can increase W (or decrease L) so that W beats L without changing the outcomes of the previous comparisons. Since the algorithm cannot see the numbers but only the comparison results, this action by the adversary is totally legal.

In the initial state, $i + j + k = n$, while in the final state $i + j + k = 2$. Since only a WW or LL comparison decreases $i + j + k$ (by 1), we need to make at least $n - 2$ WW or LL comparisons. Next consider the change of i in the initial and final states. It decreases from $i = n$ to $i = 0$. Since WW and LL comparisons never change the value of i , some NN, NW, NL comparisons are needed. Since such a comparison decreases i by at most 2, a minimum of $\lceil \frac{n}{2} \rceil$ comparisons are needed. So the total number of comparisons necessary is $n - 2 + \lceil \frac{n}{2} \rceil = \lceil \frac{3n}{2} \rceil - 2$.

6.4 Reduction

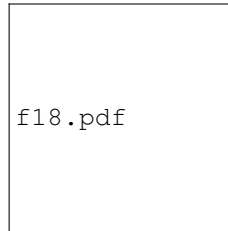
- Proving lower bounds by reduction:

Let P be the problem whose lower bound, $f(n)$, we wish to prove. Let Q be a problem whose lower bound, $g(n)$, is known. For any algorithm A for P , we try to define an algorithm B for Q which calls A as follows:



We assume that there is an algorithm, say A , for P that takes time $o(f(n))$. Then from the chart above, there is an algorithm, B , for Q , with time $t_1(n) + o(f(n)) + t_2(n)$. If $t_1(n) + o(f(n)) + t_2(n) = o(g(n))$, then we just found an algorithm which beats the proven lower bound, $g(n)$, for Q . This is of course impossible. So there is no algorithm for P with time faster than $f(n)$. Therefore, $f(n)$ is a lower bound for P .

- The convex hull problem: Given a set P of n points on the 2-D plane. The *convex hull* of the point set, denoted by $CH(P)$, is a convex polygon, represented by its vertices in counterclockwise order, say, p_1, p_2, \dots, p_k , where $p_i \in P$ for $i = 1, 2, \dots, k$, such that the polygon contains all the other points in P . See the following figure for an example.



Next we will prove that any algorithm that finds $CH(P)$ for a given P of n points takes at least $n \log n$ time in the worst case.

Assume there is an algorithm A for the convex hull problem which takes $o(n \log n)$ worst-case time. Now we will design a sorting algorithm B that calls A . For any input list of n numbers x_1, x_2, \dots, x_n , algorithm B first converts the list in $O(n)$ to a point set $(x_1, x_1^2), (x_2, x_2^2), \dots, (x_n, x_n^2)$ and feed the point set as input to algorithm A . Algorithm A then computes the convex hull of the point set in $o(n \log n)$ time. Because of the special locations of the points, the convex hull will contain all points as the vertices of the polygon, and the points are output in counterclockwise order. Algorithm B then spends $O(n)$ to search the point in the output list with the smallest x -coordinate, and from there to produce a list of n x -coordinates, which is clearly the sorted version of the original input list to B . So algorithm B sorts a list of n numbers in time $O(n) + o(n \log n) + O(n) = o(n \log n)$. This is a contradiction to the $\Omega(n \log n)$ lower bound for sorting.

f19.pdf

6.5 Trivial lower bounds by input/output

- We can also use the amount of input data that any correct algorithm needs to process or the amount of data that any correct algorithm needs to produce to establish trivial lower bounds.
- For example in the matrix multiplication problem, since the multiplication of two $n \times n$ matrices is another $n \times n$ matrix and there are n^2 entries to compute, n^2 is obviously a lower bound. (Not yet proved to be tight.)
- For another example, consider the problem of generating all permutations of n objects. Since there are $n!$ permutations to generate, $\Omega(n \cdot n!)$ is an obvious lower bound. (Tight.)
- Evaluation of an n -degree polynomial requires that all $n + 1$ coefficients be processed. Thus the trivial lower bound is $\Omega(n)$. (Tight.)
- Some bounds established by input/output are not correct lower bounds. For example, searching target in a sorted array of n elements does not require to check all n elements.
- Some lower bounds established by input/output are not trivial to prove. For example, determining connectivity of an undirected graph by its adjacency matrix requires to check the existence of each of the $n(n - 1)/2$ potential edges.

7 NP-Completeness

7.1 Optimization versus decision

- Example: Graph Coloring

Optimization: Given $G = (V, E)$, color the nodes with the minimum number of colors such that no two adjacent nodes connected by an edge have the same color.

Decision: Given $G = (V, E)$ and $B > 0$ (an integer), is there a feasible coloring of the nodes using at most B colors?

Optimization	Decision
Minimize $f(S)$	$f(S) \leq B$
Maximize $f(S)$	$f(S) \geq B$

- Theorem: If there is an algorithm with time $\Theta(T(n))$ for OPT, then there is an algorithm with time $\Theta(T(n))$ for DEC. In other words, DEC is always no harder than its OPT.

f28.pdf

7.2 The class P

- Definition: **P** is the class of problems solvable in polynomial time.

$O(n^c)$, where c is a constant.

Tractable (not so hard).

- Why use polynomial as the criterion?

If a problem is not in **P**, it will be extremely expensive and probably impossible to solve in practice for large sizes.

Polynomials have nice closure properties: $+$, $-$, $*$, and composition.

P is independent of models of computation: 1TM, kTM, RAM, etc..

7.3 The class NP

- Definition: **NP** is the class of problems solvable in polynomial time by nondeterministic algorithms.
- Definition: Nondeterministic algorithms \Leftrightarrow nondeterministic computer (a hypothetical model that doesn't exist)

– Guessing phase: Guess a solution (always on target).

– Verifying phase: Verify the solution.

- Example: Graph Coloring is in **NP**. Consider the decision problem. A nondeterministic algorithm first guesses a coloring scheme (in polynomial time) and then verify if the coloring uses no more than B colors and if any two adjacent nodes have different colors (in polynomial time).

- Theorem: $\mathbf{P} \subseteq \mathbf{NP}$.

Any ordinary (deterministic) algorithm is a special case of a nondeterministic algorithm.

- Theorem: Any $\Pi \in \mathbf{NP}$ can be solved by a deterministic algorithm in time $O(c^{p(n)})$ for some $c > 0$ and polynomial $p(n)$.
- Open problem: $\mathbf{NP} \subseteq \mathbf{P}$? or $\mathbf{P} = \mathbf{NP}$?

7.4 Polynomial reduction

- Definition: Let Π_1 and Π_2 be two decision problems, and $\{I_1\}$ and $\{I_2\}$ be sets of instances for Π_1 and Π_2 , respectively. We say there is a polynomial reduction from Π_1 to Π_2 , or $\Pi_1 \leq_p \Pi_2$ if there is $f : \{I_1\} \rightarrow \text{subset of } \{I_2\}$ such that (1) f can be computed in polynomial time and (2) I_1 has a “yes” solution if and only if $f(I_1)$ has a “yes” solution.
- Theorem: If $\Pi_1 \leq_p \Pi_2$, then $\Pi_2 \in \mathbf{P}$ implies $\Pi_1 \in \mathbf{P}$.
- Theorem: If $\Pi_1 \leq_p \Pi_2$ and $\Pi_2 \leq_p \Pi_3$, then $\Pi_1 \leq_p \Pi_3$.
- Remark: \leq_p means “no harder than”.
- Example: $\text{PARTITION} \leq_p \text{KNAPSACK}$.

PARTITION:

INSTANCE: A finite set A of numbers.

QUESTION: Is there $A' \subseteq A$ such that $\sum_{a \in A'} a = \sum_{a \in A - A'} a$?

KNAPSACK:

INSTANCE: $U = \{u_1, \dots, u_n\}$, $w : U \rightarrow \mathbb{Z}^+$, $v : U \rightarrow \mathbb{Z}^+$, and $W, V \in \mathbb{Z}^+$.

QUESTION: Is there $U' \subseteq U$ with $\sum_{u \in U'} w(u) \leq W$ such that $\sum_{u \in U'} v(u) \geq V$?

For any instance for PARTITION: $A = \{a_1, \dots, a_n\}$, let $U = \{u_1, \dots, u_n\}$ with each u_i corresponding to each a_i . Let $w(u_i) = v(u_i) = a_i$, and $W = V = \frac{1}{2} \sum_i a_i$.

This reduction can certainly be done in polynomial time. Next we need to prove that there is $A' \subseteq A$ with $\sum_{a \in A'} a = \frac{1}{2} \sum_i a_i$ if and only if there is $U' \subseteq U$ with $\sum_{u \in U'} w(u) \leq W$ and $\sum_{u \in U'} v(u) \geq V$. The proof is straightforward.

7.5 The class NPC

- Definition: **NPC** (**NP**-complete) is the class of the hardest problems in **NP**, or equivalently, $\Pi \in \mathbf{NPC}$ if $\Pi \in \mathbf{NP}$ and $\forall \Pi' \in \mathbf{NP}, \Pi' \leq_p \Pi$.
- Theorem: If $\Pi_1 \in \mathbf{NPC}$, $\Pi_2 \in \mathbf{NP}$, and $\Pi_1 \leq_p \Pi_2$, then $\Pi_2 \in \mathbf{NPC}$.
- Theorem: If $\exists \Pi \in \mathbf{NPC}$ such that $\Pi \in \mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$.
- Theorem: If $\exists \Pi \in \mathbf{NPC}$ such that $\Pi \notin \mathbf{P}$, then $\mathbf{P} \neq \mathbf{NP}$.
- Satisfiability (SAT):

INSTANCE: A boolean formula α in *CNF* with variables x_1, \dots, x_n (e.g., $\alpha = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$).

QUESTION: Is α satisfiable? (Is there a truth assignment to x_1, \dots, x_n such that α is true?)

Cook’s Theorem: $\text{SAT} \in \mathbf{NPC}$. (Need to prove (1) $\text{SAT} \in \mathbf{NP}$ and (2) $\forall \Pi \in \mathbf{NP}, \Pi \leq_p \text{SAT}$.)

7.6 NP-complete problems

- How to prove Π is **NP**-complete:
 - Show that $\Pi \in \mathbf{NP}$.
 - Choose a known **NP**-complete Π' .
 - Construct a reduction f from Π' to Π .
 - Prove that f is a polynomial reduction by showing that (1) f can be computed in polynomial time and (2) $\forall I'$ for Π' , I' is a yes-instance for Π' if and only if $f(I')$ is a yes-instance for Π .
- Seven basic **NP**-complete problems.

f29.pdf

- 3-Satisfiability (3SAT):
INSTANCE: A formula α in CNF with each clause having three literals.
QUESTION: Is α satisfiable?
- 3-Dimensional Matching (3DM):
INSTANCE: $M \subseteq X \times Y \times Z$, where X, Y, Z are disjoint and of the same size.
QUESTION: Does M contain a matching, which is $M' \subseteq M$ with $|M'| = |X|$ such that no two triples in M' agree in any coordinate?
- PARTITION:
INSTANCE: A finite set A of numbers.
QUESTION: Is there $A' \subseteq A$ such that $\sum_{a \in A'} a = \sum_{a \in A - A'} a$?
- Vertex Cover (VC):
INSTANCE: A graph $G = (V, E)$ and $0 \leq k \leq |V|$.
QUESTION: Is there a vertex cover of size $\leq k$, where a vertex cover is $V' \subseteq V$ such that $\forall (u, v) \in E$, either $u \in V'$ or $v \in V'$?
- Hamiltonian Circuit (HC):
INSTANCE: A graph $G = (V, E)$.
QUESTION: Does G have a Hamiltonian circuit, i.e., a tour that passes through each vertex exactly once?
- CLIQUE:
INSTANCE: A graph $G = (V, E)$ and $0 \leq k \leq |V|$.
QUESTION: Does G contain a clique (complete subgraph) of size $\geq k$?

7.7 Approximation algorithms

- Goal: A fast algorithm (polynomial time such as $O(n)$ and $O(n \log n)$) that produces near-optimal solution.
- Notation:
 - Π : An optimization **NP**-complete problem.
 - I : Any instance of the problem.
 - OPT : The exponential-time optimal algorithm.
 - A : A polynomial-time approximation algorithm.
 - $OPT(I)$: Optimal solution (value) for I obtained by applying OPT .
 - $A(I)$: Approximation solution (value) for I when A is used.

- How good is A ?

— Performance ratio:

$$R_A^n = \max_{I: |I|=n} \{A(I)/OPT(I)\} \text{ for minimization problems.}$$

$$R_A^n = \max_{I: |I|=n} \{OPT(I)/A(I)\} \text{ for maximization problems.}$$

$$\text{Asymptotic ratio: } R_A^\infty = \lim_{n \rightarrow \infty} \sup R_A^n.$$

$$R_A^n, R_A^\infty > 1. \text{ The smaller the better.}$$

— Relative error:

$$\epsilon_A^n = \max_{I: |I|=n} \{|A(I) - OPT(I)|/OPT(I)\}.$$

- SET COVERING

INSTANCE: A finite set X and a family F of subsets of X such that $X = \cup_{S \in F} S$.

GOAL: Determine $C \subseteq F$ with the minimum size such that $X = \cup_{S \in C} S$, i.e., members of C cover all of X .

The decision version of SET COVERING is **NP**-complete. (Reduce from VERTEX COVER.)

An example of SET COVERING: $X = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ and $F = \{S_1, S_2, S_3, S_4, S_5, S_6\}$, where $S_1 = \{1, 2, 5, 6, 9, 10\}$, $S_2 = \{6, 7, 10, 11\}$, $S_3 = \{1, 2, 3, 4\}$, $S_4 = \{3, 5, 6, 7, 8\}$, $S_5 = \{9, 10, 11, 12\}$, and $S_6 = \{4, 8\}$. The optimal solution is $C = \{S_3, S_4, S_5\}$.

An approximation algorithm based on the greedy strategy:

GREEDY-SET-COVER(X, F)

```

 $U \leftarrow X$ 
 $C \leftarrow \emptyset$ 
while  $U \neq \emptyset$  do
  select an  $S \in F$  that maximizes  $|S \cap U|$ 
   $U \leftarrow U - S$ 
   $C \leftarrow C \cup \{S\}$ 
return  $C$ 

```

Applying GREEDY-SET-COVER to the example, we have a cover C of size 4 containing S_1, S_4, S_5, S_3 .

Theorem: For any instance, GREEDY-SET-COVER finds a cover C of size no larger than $H(\max\{|S| : S \in F\})$ times the optimal size. (Note: $H(d) = \sum_{i=1}^d 1/i$ is the d th harmonic number.)

Proof. Let S_i be the i th subset selected by GREEDY-SET-COVER. We assume that the algorithm incurs a cost of 1 when it adds S_i to C . We spread this cost of selecting S_i evenly among the elements covered for the first time by S_i . Let c_x denote the cost allocated to element $x \in X$. Each element is assigned a cost only once, when it is covered for the first time. If x is covered for the first time by S_i , then

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})|}.$$

The algorithm finds a solution C of total cost $|C|$, and this cost has been spread out over the elements of X . Therefore, since the optimal cover C^* also covers X , we have

$$|C| = \sum_{x \in X} c_x \leq \sum_{S \in C^*} \sum_{x \in S} c_x.$$

The remainder of the proof rests on the following key inequality. For any $S \in F$,

$$\sum_{x \in S} c_x \leq H(|S|).$$

Suppose the above is true. We have

$$|C| \leq \sum_{S \in C^*} \sum_{x \in S} c_x \leq \sum_{S \in C^*} H(S) \leq |C^*| \cdot H(\max\{|S| : S \in F\}).$$

Now let us focus on the proof of the inequality $\sum_{x \in S} c_x \leq H(|S|)$. For any $S \in F$ and $i = 1, \dots, |C|$, let

$$u_i = |S - (S_1 \cup S_2 \cup \dots \cup S_i)|$$

be the number of elements in S remaining uncovered after S_1, \dots, S_i have been selected by the algorithm. We define $u_0 = |S|$ to be the number of elements in S , which are initially uncovered. Let k be the least index such that $u_k = 0$, so that every element in S is covered by at least one of the sets S_1, \dots, S_k . Then $u_{i-1} \geq u_i$ and $u_{i-1} - u_i$ elements of S are covered for the first time by S_i for $i = 1, \dots, k$. Thus

$$\sum_{x \in S} c_x = \sum_{i=1}^k (u_{i-1} - u_i) \frac{1}{|S_i - (S_1 \cup \dots \cup S_{i-1})|}.$$

Observe that

$$|S_i - (S_1 \cup \dots \cup S_{i-1})| \geq |S - (S_1 \cup \dots \cup S_{i-1})| = u_{i-1},$$

because the greedy choice of S_i guarantees that S cannot cover more new elements than S_i does. So we obtain

$$\sum_{x \in S} c_x \leq \sum_{i=1}^k (u_{i-1} - u_i) \frac{1}{u_{i-1}}.$$

For integer a and b with $a < b$,

$$H(b) - H(a) = \sum_{i=a+1}^b \frac{1}{i} \geq (b-a) \frac{1}{b}.$$

Using this inequality,

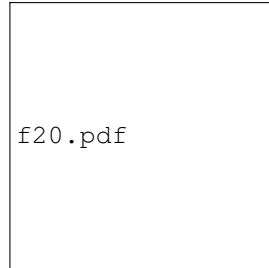
$$\sum_{x \in S} c_x \leq \sum_{i=1}^k (H(u_{i-1}) - H(u_i)) = H(u_0) - H(u_k) = H(u_0) = H(|S|).$$

Corollary: GREEDY-SET-COVER has a performance ratio of $\ln |X| + 1$.

8 Parallel Algorithms

8.1 Introduction

- The PRAM model:
Parallel Random Access Machine



Processors P_0, \dots, P_{p-1} : (1) working synchronously (at the same pace); (2) communicating through the shared memory; (3) $P_i = \text{RAM}$, $\forall i$; (4) unit-time memory access; (5) SIMD: Single-instruction, multiple-data; (6) MIMD: Multiple-instruction, multiple data.

- Concurrent versus exclusive memory accesses: EREW, CREW, ERCW, CRCW.
How to implement concurrent write: (1) common; (2) arbitrary; (3) priority; (4) combining.
- Pseudocode:

“for all $x \in X$ in parallel do $instruction(x)$ ”

Statement includes the following steps:

(1) Assign a processor to each $x \in X$. (In constant time, $P_{code(x)} \Leftrightarrow x$ and $P_i \Leftrightarrow x = code^{-1}(i)$.)

(2) Execute in parallel and by the assigned processors all those operations specified by $instruction(x)$. (The central control unit is used to synchronize.)

Example: Find the minimum of n numbers.

A is the array of n numbers and B is an auxiliary array (if $B[i] = 1$ then $A[i]$ is not the min, but if $B[i] = 0$, then $A[i]$ is the min). min stores the output.

1. for all i , $1 \leq i \leq n$, in parallel do $B[i] \leftarrow 0$
2. for all ordered pairs (i, j) , $1 \leq i, j \leq n$, in parallel do
if $A[i] < A[j]$ $B[j] \leftarrow 1$
3. for all i , $1 \leq i \leq n$, in parallel do
if $B[i] = 0$ $min \leftarrow A[i]$

Analysis: CRCW algorithm; input size is n ; the number of processors is $p = n^2$; the time is $t = \Theta(1)$. So the algorithm finds the minimum of n numbers in constant time, using n^2 processors.

- Complexity of parallel algorithms:

n : Input size;

p : The number of processors;

t : The parallel running time;

polylog: $O((\log n)^k)$ for some constant k ;

A parallel algorithm is efficient if it takes polylog time $O((\log n)^k)$ using a polynomial $O(n^c)$ number of processors;

The class **NC**: Nick (Pippenger)’s class, includes problems with efficient parallel algorithms;

A parallel algorithm is optimal if $pt = O(n)$;

P-complete problems: in **P** and no efficient parallel algorithms have been found so far;

A hypothetical picture based on **P**≠**NP** and **P**≠**NC**:

f21.pdf

- Reducing the number of processors (Brent's theorem):

If a p -processor PRAM algorithm A runs in parallel time t , then for any $p' < p$ there is a p' -processor PRAM algorithm A' for the same problem that runs in parallel time $O(pt/p')$. (Note: $pt = p't'$.)

8.2 Algorithmic technique 1: Pointer jumping

- Manipulating lists with pointers
- Example 1. List ranking.

L is a list of n elements, where $next[i]$ points to the next element following i in L . Output $rank[i]$, $\forall i \in L$, is the rank/order number of i in L . $rank[i] = 0$ if $next[i] = nil$ and $rank[i] = rank[next[i]] + 1$ if $next[i] \neq nil$.

Assign a process to each element in L .

Each node in the list has two fields: $rank$ and $next$.

f22.pdf

ListRanking(L)

```

for all  $i \in L$  in parallel do
    if  $next[i] = nil$   $rank[i] \leftarrow 0$ 
    else  $rank[i] \leftarrow 1$ 
while  $\exists i$  such that  $next[i] \neq nil$  do
    for all  $i \in L$  in parallel do
        if  $next[i] \neq nil$ 
             $rank[i] \leftarrow rank[next[i]] + rank[i]$ 
             $next[i] \leftarrow next[next[i]]$ 

```

Analysis: $p = n$ and $t = O(\log n)$.

- Example 2. Prefix computation.

Input: x_1, x_2, \dots, x_n .

Output: y_1, y_2, \dots, y_n such that $y_1 = x_1$ and $y_i = y_{i-1} \otimes x_i = x_1 \otimes \dots \otimes x_i$, where \otimes is any associative binary operator.

Define $[i, j] = x_i \otimes \dots \otimes x_j$, $1 \leq i \leq j \leq n$. Then $[i, i] = x_i$ and $[i, j] = [i, k] \otimes [k+1, j]$, $1 \leq i \leq k < j \leq n$.

We wish to compute $[1, i]$, $1 \leq i \leq n$.

f23.pdf

ListPrefix(L)

```
for all  $i \in L$  in parallel do
   $y[i] \leftarrow x[i]$ 
while  $\exists i \in L$  such that  $next[i] \neq nil$  do
  for all  $i \in L$  in parallel do
    if  $next[i] \neq nil$ 
       $y[next[i]] \leftarrow y[i] \otimes y[next[i]]$ 
       $next[i] \leftarrow next[next[i]]$ 
```

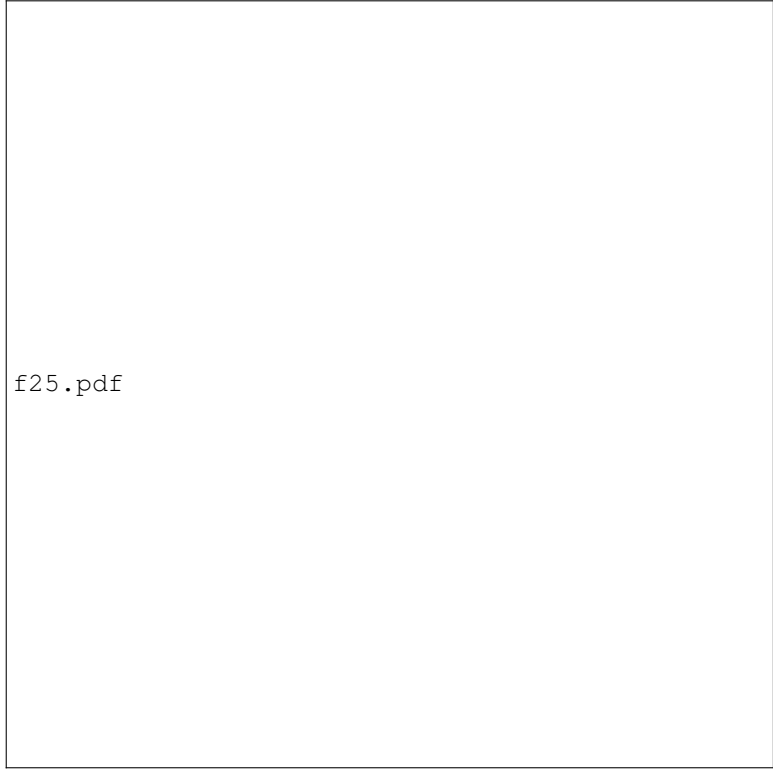
- Example 3. Computing the depth of each node in an n -node binary tree.

f24.pdf

The algorithm contains two steps. One is to construct the so-called Euler tour. The second step is to do a prefix computation, which requires $3n$ processors in $O(\log n)$ parallel time.

How to construct an Euler tour in $O(1)$ parallel time with n processors:

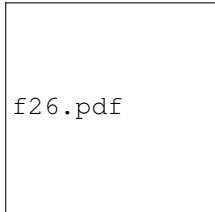
```
for all nodes  $i$  in the tree in parallel do
  if left child  $j$  exists
    make  $i.A$  to point to  $j.A$ 
    make  $j.C$  to point to  $i.B$ 
  else make  $i.A$  to point to  $i.B$ 
  if right child  $k$  exists
    make  $i.B$  to point to  $k.A$ 
    make  $k.C$  to point to  $i.C$ 
  else make  $i.B$  to point to  $i.C$ 
```



f25.pdf

8.3 Algorithmic technique 2: Balanced binary tree

- Each internal node represents the computation of a subproblem.
- The root represents the computation of the overall problem.
- Bottom-up design.
- Parallel time is at least the height of the tree.
- Example: Computing the minimum of n numbers.




f26.pdf

- $p = \frac{n}{2}$ and $t = \log n$.

8.4 Algorithmic technique 3: Divide and conquer

- Top-down design with subproblems solved in parallel.
- Parallel time is at least the depth of the recursion.
- Example: Given a polynomial $p(x)$ of degree $n = 2^k - 1$. Evaluate $p(x)$ at $x = x_0$.
 $p(x) = x^{\frac{n+1}{2}} q(x) + r(x)$, where $\deg(q) = 2^{k-1} - 1$ and $\deg(r) \leq 2^{k-1} - 1$.



f27.pdf

Depth is $O(\log n)$. So is the parallel time.

8.5 Algorithmic technique 4: Compression and collapsing

- Example: Finding the minimum.

$A[i]$ and $A[i+1]$ is compressed to $\min\{A[i], A[i+1]\}$ for odd i .

$[3, 7, 8, 3, 9, 2, 3, 1] \Rightarrow [3, 3, 2, 1] \Rightarrow [3, 1] \Rightarrow [1]$.

- Example: Graph algorithms: Compress a graph recursively into a single (super-)vertex.
- Example: Euler tour: Collapse a node into three subnodes.