Recursive Algorithm :

"An algorithm to solve a problem by repeatedly 'calling itself' with subproblems.'

Ex) : mergesort, quicksort ...

Methods for Recurrences :

1. Iteration

T(n) = n + 3T(n/4)

= n + 3(n/4 + 3T(n/16)) = n + 3n/4 + 9T(n/16)

= n + 3n/4 + 9(n/16 + 3T(n/64)) = n + 3n/4 + 9n/16 + 27T(n/64)

= ...

= n + (3/4)n + (9/16)n + ... + (3^(log_4 n))T(n/(4^(log_4 n))))

= n + (3/4)n + (9/16)n + ... + (3^(log_4 n))T(1)

<= n * (sigma i=0 to inf){(3/4)^i} + n^(log_n 3) * THETA(1)

= 4^n + o(n)

= THETA(n)

Therefore, T(n) = O(n). QED.

2. Guess & Verification

This method uses mathematical induction.

Ex)

T(n) = 2T(n/2) + n, T(1) = 1

Let's guess that T(n) = O(n * (log n)).

i.e. T(n) <= c * n * (log n) for some c > 0 and large enough n.

Pf.

Base case :

T(2) = 2T(1) + 2

= 4

<= c * 2 * (log 2)

c for above inequality exists.

Inductive case :

Assume that for arbitary k < n, there exists c > 0 s.t. T(k) <= c * k * (log k).

Then,

T(n) = 2T(n/2) + n

<= 2 * c * n/2 * (log n/2) + n

= c * n * (log n) - c * n * (log 2) + n

Here, if we choose c large enough, -c * n * (log 2) + n becomes negative.

And that holds for c >= 1.

Therefore, T(n) <= c * n * (log n) for c >= 1

We choose c = 2, n_0 = 2 (n_0 : large enough n)

* We can always verify any claim for a small fixed n.(e.g. n=2)

e.g. T(n) = 10 * T(n/10) + n, T(1) = 1

T(n) >= c * n * (log n)

T(10) = 10 * T(1) + 10 = 20 <= c * 10 * (log 10) (there exists c that satisfies this.)

Thus, we don't have to explicitly prove the boundary case in 'Guess & Verification' methods.

EX)

T(n) = 2 * T(n/2 + 17) + n

Let's guess that T(n) = O(n * (log n)).

i.e. T(n) <= c * n * (log n) for some c > 0 and large enough n.

Pf.

T(n) = 2 * T(n/2 + 17) + n

<= 2 * c * (n/2 + 17) * (log(n/2 + 17)) + n

= c(n + 34) * (log(n/2 + 17)) + n

<= c(n + 34) * (log (3n/4)) + n

(This assumption holds when n/2 + 17 <= 3n/4, n > 68)

= c * n * (log n) + c * n * (log 3n/4) + 34c(log (3n/4)) + n

= c * n * (log n) + n * ((c * log(3/4)) + 1) + 34c(log 3n/4)

<= c * n * (log n)

(Last inequality holds when n * ((c * log(3/4)) + 1) goes to negative.

And that holds for c >= (1 / (log 4/3)).)

Above inequality holds for sufficiently large n. (choose c = 5)

CAUTION : Do Not Introduce new constant such as c'.

3. Master Theorem

Given a recurrence relation such as

T(1) = 1 AND T(n) = a * T(n/b) + f(n) (for n > 1, f(n) is polynomial function of n.)

(Where a, b are positive constants, f(n) = O(g(n)) for some polynomial fuction g(n).)

T(n) = f(n) + a * T(n/b)

= f(n) + a * (f(n/b) + a * T(n/(b^2)))

= f(n) + a * (f(n/b) + a * (f(n/(b^2)) + a * T(n/(b^3))))

= ...

= (sigma i=0 to k-1)(a^i * f(n/b^i)) + a^k * T(n/b^k)

Assume n = b^k

T(n) = (sigma i=0 to k-1)(a^i * f(n/b^i)) + n^(log_b a)

= (Particular solution) + (homogeneous solution)

Homogeneous solution is a key standard for determining the running time of given algorithm.

This solution is the cost of solving the boundary subproblems of case 1.

Particular solution is the sum of overheads.

Master Theorem :

Given a recurrence relation of

T(1) = c AND T(n) = a * T(n/b) + f(n) for n > 1.

(When a >= 1, b > 1 and c > 0 are constants,

f(n) = O(g(n)) for some polynomial function g(n)).

Let the homogeneous function(solution) to be n^(log_b a) = h(n)

Case 1 :

If f(n)/h(n) = O(1/(n^epsilon)) for some epsilon > 0,

(Homogeneous function overwhelms overhead function.)

then T(n) = THETA(h(n))

Case 2 :

If f(n)/h(n) = O((log n)^k) for some non-negative integer k,

then T(n) = THETA(h(n) * (log n)^(k + 1))

Case 3 :

If f(n)/h(n) = OMEGA(n^epsilon) for some epsilon > 0

and a * f(n/b) < f(n) for sufficiently large n,
(Original problem's overhead dominates next-upper problem's overhead.)
then T(n) = THETA(f(n))

Ex :

T(n) = 3 * T(n/2) + n
a = 3, b = 2, f(n = n, h(n) = n ^ (log_2 3)
f(n)/h(n) = n / (n ^ (log_2 3)) = n ^ (1 - log_2 3) = O(n^(1 - log_2 3))
Therefore, T(n) = THETA(h(n)) = THETA(n^(log_2 3))

T(n) = T(n/2) + d
a = 1, b = 2, f(n) = d, h(n) = n^(log_2 1) = 1
f(n)/h(n) = d = THEGA((log_2 n)^0)
Therefore, T(n) = THETA(log n)

T(n) = T(n/2) + (log n)
a = 1, b = 2, f(n) = log n, h(n) =
f(n)/h(n) = (log n)/1 = THETA(log n)
Therefore, T(n) = THETA((log n)^2)

T(n) = T(n/2) + n
a = 1, b = 2, f(n) = n, h(n) = n^(log_2 1) = 1
f(n)/h(n) = n/1 = n = SIGMA(n^1) (epsilon = 1)
1 * f(n/2) = n/2 < n = f(n)
Therefore, T(n) = THETA(f(n)) = THETA(n)

Cf. Master Theorem holds for every natural number n. It does not have to be in form of n = 2^k
Without Loss of Generality, assume n = 2^k, 4^k, ...
and T(n) is non-decreasing function.
For 2^k, 2^(k+1), we can find that f((2n)^p) = f(2^p * n^p) and 2^p will be constant.
So, for enough large n, in asymptotic aspect, we can neglect 2^p and this will give us
2^k and 2^(k+1) are similar.

Others :

Changing variables -
T(n) = 2T(|_sqrt(n)_|) + (log n)

Now let n = 2^m

T(2^m) = 2T(2^(m/2)) + m

And rename T(2^m) = S(m). Then we obtain

S(m) = 2 * (m/2) + m

Now we can use Master Theorem.

By Master Theorem, S(m) = THETA(m * (log m))

Therefore, T(n) = THETA((log n) * (log log n)).

Using a stronger inductive hypothesis -

T(n) = 2 * T(n/2) + 1

Guess that T(n) <= c * n for enough large n.

T(n) <= 2 * c * (n/2) + 1 = c * n + 1

This is not LEQ than c * n.

Instead, guess T(n) <= c * n - 2

T(n) <= 2 * (c * n/2 - 2) + 1 = c * n - 3 <= c * n – 2.

Sorting and Order Statistics

Quicksort :

Pseudo Code :

```
Quicksort(A[], p, q)
    if p < q then
        r <- partition(A, p, q)
        Quicksort(A, p, r-1)
        Quicksort(A, r+1, q)
partition (A[], p, q)
    x <- A[p] // pivot
    last S_1 <- p // set S_1 and S_2 are empty.
    for i <- p+1 to q do // i : index of the 1st unknown
        if (A[i] < x) then
            Last S_1++;
            A[Last S_1] <-> A[i] // swap.
    A[Last S_1] <-> A[p]
    return Last S_1
```

Quicksort divides itself to two smaller part. So if that smaller problems return correct(sorted) results,
overall sorting will return correct result.
Partition function needs THETA(n) time.
Cf. The Partition function does not guarantee a balanced partition.
It depends on pivot.
Worst-Case running time :

1) $T(n) = \max_{0 \le k \le n-1}\{T(k) + T(n - k - 1)\} + \Theta(n)$ (max function is used due to worst-case analysis).

$\ge T(0) + T(n-1) + \Theta(n)$

$= T(n-1) + \Theta(n)$

$= \sum_{k=1 \text{ to } n}\{\Theta(k)\}$

$= \Theta(\sum_{k=1 \text{ to } n}\{k\})$

$= \Theta(n^2)$

Therefore, T(n) = SIGMA(n^2)

2) Guess T(n) = O(n^2).

In other words, T(n) <= c * (n^2) for some c > 0.

T(n) = max_(k=0 to n-1){T(k) + T(n - k - 1)} + THETA(n)

<= max_(k=0 to n-1){c*(k^2) + c(n - k - 1)^2} + THETA(n)

= c * (max_(k=0 to n-1){k^2 + (n - k - 1)^2}) + THETA(n)

= c * n^2 - 2cn + c + THETA(n)

<= c * n^2 for some c and large enough n such that 2cn dominates c + THETA(n).

Therefore, T(n) = O(n^2).

From above 1 and 2, T(n) = THETA(n^2)

Average-Case running time :

Cf. If always well balanced,

T(n) = 2 * T(n/2) + THETA(n)

By Master Theorem, T(n) = THETA(n * (log n)).

Cf. If always partitioned to 1 : 2?

T(n) = T(n/3) + T(2n/3) + THETA(n)

Result is same with Average-Case.

Assume every input permutation occurs with an equal probatility.

All possible cases are n!.

Assume all keys are distinct.

Therefore, T(1) = THETA(1).

T(n) = (1/n)sigma_(i=0 to n-1){T(i) + T(n - i - 1)} + THETA(n) (This means pivot is (i+1)th element.)

= (2/n)sigma_(i=0 to n-1){T(i)} + THETA(n)

Guess T(n) <= c * n(log n) for some c > 0.

Then, T(n) = (2/n)sigma_(k=0 to n-1){T(k)} + THETA(n)

<= (2/n)sigma_(k=2 to n-1){T(k)} + THETA(n) (THETA(n) absorbed T(0), T(1).)

<= (2/n)sigma_(k=2 to n-1){c * k * (log k)} + THETA(n)

= (2c/n)sigma_(k=2 to n-1){k * (log k)} + THETA(n)

<= (2c/n)integral_(1 to n){x * (log x)}dx + THETA(n)

= (2c/n)([1/2 * x^2 * (log x)]_(1 to n) - [x^2 / 4]_(1 to n)) + THETA(n)

= c * n * (log n) + (-cn/2 + c/2n + THETA(n))

<= c * n * (log n) (We can choose c > 0 such that cn/2 dominated THETA(n).)

Therefore, T(n) = O(n * (log n)).

Randomized Quicksort :

It is a quicksort that pivot element is not effected by input list.

Pivot is selected after input list is determined.

Running time is the same as quicksort.

Decision-Tree methods of computation.

Assume all elements for sorting are distinct.

Then, Every sorting algorithm that is based on comparison has time complexity of SIGMA(n * (log n)).

Def. A decision tree is a binary tree in which the internal nodes are labeled with a comparison of the form a_i : a_j and the edges are labeled with "<" or ">"

On an input sequence(a_1, a_2, ..., a_n) the computation of a decision tree starts at the root.

Suppose we are at node n with labeled a_i : a_j, then a_i and a_j are compared with each other.

If a_i > a_j then move to the right child of n, else move to the left child of n.

The computation stops when we reach a leaf node.

Ex) A decision tree for sorting 3 numbers a, b, c

This decieion tree has 6 possible outputs.(For n numbers : n!)

Height of this tree is worst-case running time of decision.

Fact 1 :

A decision tree that sorts n numbers must have n! leaves.

Fact 2 :

A binary tree of height h have at most 2^h leaves.

Theorem :

A decision tree for sorting n numbers must have height at least (log_2 (n!)).

Pf. Straight forward from Fact 1 and Fact 2.

Corollary :

Any comparison-based sorting algorithm has to take time SIGMA(n * (log n)) in the worst case.

Pf.

n! = sqrt(2 * pi * n) * (n/e)^n * (1 + THETA(1/n)) (From Stirling's approximation.)
> (n/e)^n
We take log of base 2 to both side.
log (n!) > n * (log n) - n * (log e) = THETA(n * (log n)).

Linear-Time Sortings :

Note.

The lower bound SIGMA(n * (log n)) is for sortings based on comparisons.
However, when some conditions are met, lower-than-O(n * (log n))
sortings are possible.

Counting sort :

when each input is an integer in [1, O(n)], THETA(n)-time sorting is possible.
Pseudo-Code :

```
Counting-Sort(A[], n, k)
// n : Number of elements
// k : A[i] in [1, k], forall i = 1, ..., n
        for i in [1 .. k]
                C[i] <- 0
        for i in [1 .. n]
                C[A[i]]++
        for i in [1 .. k]
                C[i] <- C[i] + C[i-1]
        // Here, C[i] : Number of elements not greater than i.
        for i in [n .. 1]
                B[C[A[i]]] <- A[i]
                C[A[i]]--
```

Running time is THETA(n) obviously.

Radix sort :

Pseudo-Code :

```
Radix-Sort(A[], d, n) =
// d : Number of digits in A[i] for all i = 1, ..., n
```

for i in [1 .. d]

sort A[1 .. n] on digit i by a stable sort.(This sort should finish in linear time.)

Running Time : THETA(k * n) = THETA(n) (k : constant).

Bucket sort :

Pseudo-Code :

Bucket-Sort(A[], n) =

// A[] : array A[1 .. n] with uniform distribution in [0, 1).

for i in [1 .. n]

Insert A[i] into list B[|_ n * A[i] _|]

for i in [0 .. n-1]

Sort list B[i] using an appropriate sorting algorithm

print the list B[0], ..., B[n-1] in order.

*It does not have to be [0, 1). Transformation can be taken.

Running time(with uniform distribution) :

The expected numbers of elements n_i in each list B[i] is np=1 with variance sigma^2 = np(1-p) = 1-(1/n). (p = 1/n)

sigma^2 = E(n_i ^2) - (E(n_i))^2

Therefore, E((n_i)^2) = (1 - (1/n))^2 - 1^2 = 2 - (1/n) = THETA(1)

Every insertion sort takes O((n_i)^2) = THETA(1)

Therefore, O(n) in total for bucket sort.

Order Statistics :

Given a list of n elements,

Selecting minimum : needs n-1 comparisons.

Selecting maximum : needs n-1 comparisons.

Selecting i(th) smallest element :

A trivial upper bound is O(n * (log n)) (by sorting)

A trivial lower bound is SIGMA(n)

Average-Case O(n) algorithm :

Select(A[], p, r, i)

```
// Return i(th) smallest from A[p .. r]
        if i > r - p + 1 or i <= 0 then
                return "error!"
        if p = r then
                return A[p] // i = 1
        q <- partition(A, p, r)
        k <- q - p + 1
        if i = k then return A[q]
        else if (i < k) then select(A, p, q-1, i)
        else select(A, q+1, r, i-k)
```

Worst-Case :

T(n) = THETA(n) + max_(1 <= k <= n-1){T(k)}
>= THETA(n) + T(n-1)
>= THETA(n) + THETA(n-1) + T(n-2)
>= THETA(sigma_(i=2 to n){i}) + T(1)
= THETA(n^2)

Therefore, T(n) = SIGMA(n^2)

Average-Case :

T(n) <= THETA(n) + (1/n) * (sigma_(k=1 to n){T(max(k-1, nk))})
<= THETA(n) + (2/n) * (sigma_(k=ceiling(n/2) to n){T(k-1)})
= THETA(n) + (2/n) * (sigma_(k=ceiling(n/2)-1 to n-1){T(k)})

Now, guess T(n) <= cn for some c>0 and large-enough n.

T(n) <= THETA(n) + (2/n) * (sigma_(k=ceiling(n/2)-1 to n-1){ck})
= THETA(n) + (2c/n) * {(sigma_(k=1 to n-1){k}) - (sigma_(k=1 to ceiling(n/2)-1){k})}
<= THETA(n) + (2c/n) * ((n^2 - n)/2 - ((n/2)^2 - ...)/2)
= cn - c - (cn/4) + THETA(n)
<= cn

We can choose c>0 s.t. cn/4 dominates THETA(n) - c.

Therefore, T(n) = O(n).

(T(n) is definitely SIGMA(n), because we have to at least see n elements 1 time.)

Worst-Case O(n) algorithm :

Idea :

1) Devide the input into n/5 groups each having 5 elements.
2) Find the medians of each group.
   Let the medians be m_1, m_2, ..., m_(ceiling(n/5)).
3) Let median of m_1, m_2, ..., m_(ceiling(n/5)) be M.
4) Use M as the pivot element and partition the whole elements as in Quicksort.
5) Select the proper side of the partitions and recursively conduct steps 1~5.

(Number of elements that are equal to M or less) >= 3(n/10 - 1) = (3n/10) - 3

(Number of elements that are equal to M or greater) >= 3(n/10 - 1) = (3n/10) - 3

```
Linear-Select(A[], p, r, i) =
        if (p = r) then
                return A[p]
        Find the pivot M as described in Steps 1~5.
        Exchange A[p] and M
        q <- partition(A, p, r)
        k <- q - 1 +
        if (i = k) then
                return A[q]
        else if (i < k) then
                Linear-Select(A, p, q-1, i)
        else
                Linear-Select(A, q+1, r, i-k)
```

Worst-Case running time :

```
T(n) <= THETA(n) + THETA(n) + T(ceiling(n/2)) + THETA(n) + T((7n/10) + 3)
                // T((7n/10) + 3) is the number of remaining elements from equation n - ((3n/10) - 3)
                // First THETA(n) : step 1
```

// Second THETA(n) : step 2
// Third T(ceiling(n/2)) : step 3
// Fourth THETA(n) : step 4
// Fifth T((7n/10) + 3) : step 5 (due to (7n/10) + 3 > (3n/10) - 3).
<= THETA(n) + T((n/5) + 1) + T((7n/10) + 3)

Guess T(n) <= cn for some c > 0, n_0 > 0 and n >= n_0.
Therefore,
T(n) <= THETA(n) + c((n/5) + 1) + c((7n/10) + 3)
= c(9n/10) + 4c + THETA(n)
= cn - (cn/10) + 4c + THETA(n)
<= cn
We choose c to make (cn/10) dominates 4c + THETA(n).
Therefore, T(n) = O(n).

Dynamic Programming

Elements of dynamic programming :

Optional substructure :

an optional solution contains within it optimal solutions to the problems.

Overlapping Subproblems :

A recursive algorithm revisits the same subproblems over and over again.

Matrix-chain Multiplication :

Given a matrix <A_1, A_2, ..., A_n>, we want to compute the product A_1*A_2*...*A_n.

Note that Matrix multiplication is associative.

If A is p*q matrix and B is q*r matrix,

the cost of computing the product AB is pqr.

Object :

We want to find a sequence of multiplications with

the minimum cost.

This is finding an optimial parenthesizing.

Ex)

A : 10 * 100 matrix

B : 100 * 5 matrix

C : 5 * 50 matrix

(AB)C -> 7500 multiplications.

A(BC) -> 75000 multiplications.

Let matrix A_k be matrix of dimension p_(k-1) * p_k

We consider n matrices, A_1 to A_n.

Then we have n-1 choices, when parenthesizing into 2 parts.

General form : (A_1 * A_2 * ... * A_k)(A_(k+1) * ... A_n)

Let C_ij : Minimal computation cost of A_i * ... * A_j

Then our goal is to find C_1n

For general form described above, cost is (p_0 * p_k * p_n) + C_1k + C_(k+1)n.

So, C_1n = min_(k=1 to n-1){(p_0 * p_k * p_n) + C_1k + C_(k+1)n}.
Therefore, C_ij = min_(k=i to j-1){(p_i * p_k * p_j) + C_ik + C_(k+1)j}.
C_ij = 0 (when i=j)
min_(k=i to j-1){(p_i * p_k * p_j) + C_ik + C_(k+1)j} (when i < j)
Algorithm :

```
MC(i, j) =
// Return the minimum cost for Matrix multiplication A_i * ... * A_j
    if i=j then
        return 0
    min <- inf
    for k=i to j-1 do
        q <- MC(i, k) + MC(k+1, j) + p_(i-1)*p_k*p_j + c_ik + c_(k+1)j
        if q < min then
            min <- q
    return min
```

Running time of MC(1, n) by the number of calculations p_(i-1)*p_k*p_j :

T(n) = 0 (when n = 1)
sigma_(k=1 to n-1){T(k) + T(n-k) + 1} (when n <> 1)
= 0 (when n = 1)
2 * sigma_(k=1 to n-1){T(k) + (n-1)} (when n <> 1)

T(n) grows exponentially for n.

Guess T(n) >= c * 2^n

T(n) = 2 * sigma_(k=1 to n-1){T(k) + (n-1)}
>= 2 * sigma_(k=1 to n-1){c * 2^k + (n-1)}
= 2c(2^n - 2) + (n - 1)
= 2c * 2^n - 4c + n - 1
> c * 2^n

for large-enough n.

Therefore, T(n) = SIGMA(2^n)

Observation :

The recursive algorithm MC(i, j) calls the same subproblems repeatly.

Memorizing the minimum costs of subproblems can dramatically save running time.

Dynamic programming solution :

Construct a two-dimensional array m[][] where m[i][j] has the minimum cost for A_i * ... * A_j.

A sample bottom-up implementation :

```
for i=1 to n do
        m[i][i] = 0
for l=1 to n-1 do // l : length
        for i=1 to n-l do // i, j : <A_i * ... * A_j>
                j <- i + l
                m[i][j] <- min_(k=i to j-1){m[i][k] + m[k+1][j] + p_(i-1)*p_k*p_j}
return m[1][n]
```

Running time : THETA(n^3)

Top-down implementation is also possible :

Instead of calling the same subproblems, look at the table entries and get the values.

Called "Memoization"

```
MC(i, j) = // Assume all m[][]'s initialized to -1.
        if m[i][j] <> -1 then
                return m[i][j]
        else if i=j then
                m[i][j] <- 0
                return m[i][j]
        else
                the same as before. (in SIGMA(2^n) algorithm)
        m[i][j] <- min
        return m[i][j]
```

Running time : THETA(n^3)

Longest Common Subsequence (LCS) :

Given two sequences X=<x_1, x_2, ..., x_n> and Y=<y_1, y_2, ..., y_m>
we want to find an LCS of X and Y.
Note 1)
<BCDB> is a subsequence of <ABCBDAB>
X = <ABCBDAB>, Y = <BDCABA>
<BCA> is a common subsequence of X and Y.
<BCBA>, <BDAB> are LCSs of X and Y.

Optimal substructure of LCS :

X_m = <x_1, ..., x_m>, Y_m = <y_1, ..., y_m>
Let Z_k = <z_1, ..., z_k>
1. If x_m = y_n then z_k = x_m = y_n and z_(k-1) is an LCS of X_(m-1) and Y_(n-1)
2. If x_m <> y_n then z_k = max{LCS(X_m, y_(n-1)), LCS(X_(m-1), Y_n)}

c_ij = the length of LCS(x_i, Y_j)
c_ij = 0 (if i = 0 or j = 0)
c_(i-1)(j-1) + 1 (if i, j > 0 and x_i = y_j)
max{c_(i-1)j, c_i(j-1)} (if i, j > 0 and x_i <> y_j)
Obviously, exponential-time impelmentation is possible by a recursive algorithm.
But, because of the existance of THETA(mn) subproblems, PP is possible.
LCS-length(X, Y) =

```
//m, n : the length of sequences X and Y
for i = 0 to m do
        c[i, 0] = 0;
for j = 0 to n do
        c[0, j] = 0;
for i=1 to m do
        for j=1 to n do
                if x_i = y_j then
```

```
                    c[i, j] = c[i-1, j-1] + 1;
            else
                    c[i, j] = max{c[i-1, j], c[i, j-1]}
    return c[m, n]
```

Optimal binary search tree (static) :

1. S = {x_1, x_2, ..., x_n} where x_1 < x_2 < ... < x_n
2. P_i = the probability that member(S, x_i) is called, i = 1, ..., n
3. q_i : the probability that member(S, x) is called for x_i < x < x_(i+1), i = 0, 1, ..., n (unsuccessful search)
   (let x_0 = -inf, x_(n+1) = inf)

We want to find a BST that has the minimum expected number of comparisons.

The cost of a BST with x_1 < x_2 < .. < x_n is

SIGMA_{i=1 to n}(p_i * (depth(x_i) + D)} + SIGMA_{i=0 to n}(q_i * depth(e_i)}

c_ij = the optimal cost for a tree with x_i, x_(i+1), ..., x_j

w_ij = SIGMA_{l=i to j){p_i} + SIGMA_(l = i-1 to j){q_i}

Optimal substructure :

c_ij = 0 (if i > j)

min_(k=i to j){c_i(k-1), c_(k+1)j} + w_ij (if i <= j)

Running time for c_1n = THETA(n^3) (due to SIGMA_(i=1 to n){SIGMA_(j=i to n){(j-1+1)}} = THETA(n^3))

Static Data = Does not change once given.

Dynamic Data = Changes over time.

We want to support the following operations :

1) Member(S, x)
2) Insert(S, x)
3) Delete(S, x)
4) Min(S), Max(S)
5) Union(A, B, C) (C = A U B)
6) Find(x)

Data structures that supports :

Member, Insert, Delete : Dictionary
Min, Insert, Delete : Priority Queue
Uniun, Find : UnionFind

Hash Table : Can be used for dictionary
BST : Can be used for both dictionary and priority queue.
Heap : Can be used for priority queue.

Hash Table :

Idea :

Hash function h, universe U, hash table size of m.
h : U -> {0, 1, 2, ..., m-1}
cf. Usually m << |U|

Collision :

If x <> y and h(x) = h(y) then we have a collision.

Hash Table is the most ideal data structure for searching data. It gives data in constant time.

Hash functions :

Assume that all keys are natural numbers.
(Non-natural number can be easily transformed to natural nmbers.)
Division method :

h(x) = x mod m

Multiplication method :

h(x) = |_ m(xA mod 1) _|, 0 < A < 1

Collision Resolution :

1) Chaining : Use linked list when same hash function value occurs.

Let m be the number of keys in a hash table.

Define the load factor alpha = n/m

Assumption :

h is computable in O(1) time.

h is distributes keys uniformly in the table.

All elements of U occur with equal probability.

Fact :

Under the above asumption, a search takes THETA(max(1, alpha)) on average.

2) Open Addressing :

h_0, h_1, ..., h_(n-1)

Insert(S, n) =

h_0(x)

if h_0(x) is occupied, h_1(x)

if h_1(x) is occupied, h_2(x)

...

if h_i(x) is occupied, h_(i+1)(x).

Ideally want h_0(x), h_1(x), ..., h_(m-1)(x) be a permutation of 0, 1, ..., m-1.

cf. Linear probing : h_i(x) = (h_0(x) + (c*i)) mod m.

cf. Quadratic probing : h_i(x) = (h_0(x) + (c_1 * i) + (c_2 * i^2)) mod m

cf. double hashing :

h, h' : hash functions.

h_i(x) = (h(x) + i * h'(x)) mod m.

Double hashing prevents secondary clustering problem of Linear or Quadratic hashing by adding one more hash function h'.

Uniform hashing :

For any key x, the probe sequence h_0(x), h_1(x), ..., h_(m-1)(x) is

a random permutation of {0, 1, ..., m-1}

This is most ideal hashing.

Theorem :

Assume uniform hashing, the expected number of probes in an unsuccessful search or an insertion is at most 1/(1-alpha), where alpha is the load factor. (At most times, 1/(1-alpha) <= 2.)

Proof :

cf. Insertion and unsuccessful search is same, since insertion occurs right after unsuccessful search.

Let p_i = Pr(exactly i probes access occupied slots)

Let q_i = Pr(at least i probes access occupied slots)

Expected number of probes = 1 + SIGMA_(all i >= 0){i * p_i} (note that p_0 = 0)

= 1 + SIGMA_(all i >= 1){q_i - q_(i+1)}

= 1 + SIGMA_(all i >= 1){q_i}

<= 1 + SIGMA_(all i >= 1){alpha^i}

Here, q_1 = n/m, q_2 = (n/m) * (n-1/m-1), ..., q_i = (n/m) * (n-1/m-1) * ... * (n-i+1/m-i+1).

n/m = alpha, n-1/m-1 > alpha, and so on.

So above inequality holds.

= 1/(1-alpha)

Theorem :

Under the same assumption the expected number of probes in a successful search is at most (1/alpha)*ln(1/(1-alpha)).

Proof :

Successful search exactly steps the same footprint of insert operation.

The load factor right after the ith key has been inserted was i/m.

If x was the (i+1)th key inserted, then the expected number of probes in a successful search for x is by the previous theorem, at most 1/(1-(i/m)) = m/(m-i)

Average overall keys.

(1/n)SIGMA_(i=0 to n-1){m/(m-i)} = (m/n)SIGMA_(i=0 to n-1){1/(m-i)}

<= (1/alpha)INTEGRAL_(0 to n){1-(m-x)}dx

= (1/alpha)*ln(1/(1-alpha))

Binary Search Tree(BST) :

All keys should have different values.

All values of node that is on left subtree is smaller than given node's value, vice versa.

There are n! sequence when size of n input is given.

We cannot guarantee the balancedness of BST when we do not know the root of BST.

When median is chosen for root of every subtree and the whole BST, BST is guaranteed to be balanced.

Theorem :

A sequence of n inputs into an empty BST takes time O(nlogn) on average.

(Assume that every permutation of the input sequence is equally likely.)

Proof :

Assume the input sequence is a_1, a_2, ..., a_n.

Let b_1, b_2, ..., b_n be the input sequence sorted in order.

a_1 will become the root of the resulting BST.

Assume a_1 = b_j for some j (1 <= j <= n).

Then, b_1, ..., b_(j-1) will go to left subtree of a_1(root).

And likely, b_(j+1), ..., b_n will go to right subtree of a_1(root).

Let T(n) be the expected number of comparisons for inserting n keys into an empty BST.

T(n) = (1/n)SIGMA_(j=1 to n){(j-1) + T(j-1) + (n-j) + T(n-j)} // j-1, n-j corresponds to comparison with root.

= (1/n)SIGMA_(j=1 to n){T(j-1) + T(n-j)} + THETA(n)

= (2/n)SIGMA_(j=1 to n-1){T(j)} + THETA(n) // This form of formula has seen with quicksort.

= T(n) = O(nlogn).

Note that worst case has THETA(n^2) time complexity.

Balanced Binary Search Tree :

(Red-Black tree as an example)

Definition : https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

(From PPT of Lecture) :

Every node in the search tree has a color : red or black.

It has to satisfy the following properties(red-black properties) :

1) Root is black

2) Every leaf is black

3) If a node is red, its children should be black

4) In any path from the root to a leaf, the nmber of black node on the path is the same. (which is called "Black height")

cf. Here, leaf does not mean ordinary leaf node. We assumes that very NIL pointer points to "NIL" leaf node.

cf. In worst case, the longest path length cannot exceed twice length of shortest path.

Threorem :

In a red-black tree of n nodes, the worst-case depth is O(log n)

Def. Black height :

The number of black nodes from, but not including, the root to a (any) leaf

Lemma :

A red-black tree with n internal nodes has height at most 2 * log(n+1).

Proof :

For any node r in a red-black tree, the subtree rooted at r contains at least 2^(BH(T)) - 1 internal nodes. --- (1)

Let h be height of the tree.

Then by property 3,

h <= 2*BH(T)

h/2 <= BH(T) --- (2)

n >= 2^(BH(T)) - 1 --- from (1)

>= 2^(h/2) - 1 --- from (2)

Therefore, h <= 2 * log(n+1)

Insertion in Red-Black tree : Refer to PPT.

Deletion in Red-Black tree : Refer to PPT.

Disjoint Sets :

Want to handle a collection of disjoint sets.

(Example usage : Kruskal algorithm)

Operations :

1) Make-Set(x) - Create a singleton {x}.

2) Union(A, B) - Create the set A OR B. Destroy A, B.

3) Find(x) - Return the name(the representative) of the set containing x.

Linked-list representation :

Represent each set by a linked list.

EX : {2, 4, 7} with representative as 2. 4->2, 7->2 (2->2. 2->next : 4, 4->next : 7)

Make-Set(x) - THETA(1)

Find(x) - THETA(1)

Union(A, B) - O(|A| + |B|) (set new representative, link representative's set's last element's next to non-representative's head.)

Weighted-Union Heuristic

- When Union(), attach the smaller set to the larger one.

Theorem :

Using linked-list representation and weighted-union heuristic,
a sequence of m make-Set, Union, and Find operations. n of which are Make-Set operations (i.e. n elements)
takes O(m + n*(logn)) time.

Proof.

<Amortized Analysis>

For element x, when changing link in x occurs n times,
set containing x has size at least 2^(k-1) (self linking counts.)
But 2^(k-1) <= n since there are n elements.
Therefore, k <= log n. and for all n elements, less than or equal to n*(log n)
So, running time is O(m + n*(log n)).
Note. The domainating cost for union is spent for updating the pointers to the representative.
For any element x, each time x's pointer is updated(in union) it belongs to the
smaller set(if it belongs to the larger set, no update needed.)
The set size containing x : 1 -> >=2 -> >=4 -> ... -> >=2^k
If the number of elements is n, there can be at most logn updates for any x.
Therefore, the total time for update is O(nlogn)
Since each Make-Set and Fin takes O(1) time, the total cost for the entire sequence is O(m + n*(log n)) = O(max{m, n*(log n)})
The amorized cost per union is O(log n)

Tree representation :

Each set is represented by a tree.

Make-Set(x) : x's link points to x.

Union(A, B) : B's root's link points to A's root.

Make the root of one tree be a child of the root of the other tree.

Find(x) : follow the parent pointers to the root. - O(n)

cf. This case, Find(x) operation is the time-dominant operation.

Two heuristics to improve running time :

1) Union by rank (almost same as weighted union.)

Each element x has an integer value rank[x], to maintain an upper bound on the height of x('s subtree).

After Make-Set(x), rank[x] = 0

After that, rank[x] changes only when rank[x] = rank[y] and y's tree is merged into x. (increased by 1)

The tree with smaller height becomes a subtree of the other tree.

Fact. With Union by rank, a sequence of n Make-Set, Union, Find takes time O(nlogn)

2) Path Compression

Each time we do a Find(x), make every node on the path from x to the root a child of the root.

Fact, With path compression(only), a sequence of n Make-Set, Union, Find takes time O(nlogn).

Theorem :

With union-by-rank and path compression, a sequence of n Make-Set, Union, Find takes O(n * (log* n)), where log* n = min{k(log log ... log n <= 1, log for k times.)}

Greedy Algorithms

Greedy Algorithm :

Choose a sequence of locally best choices.
Usually doesn't guarantee the global optimum.
cf. Examples not guaranteeing optimum : coin change, knapsack
(knapsack : filling knapsack with different value items, each may have different volumes.)
(real-number knapsack problem's optimal solution can be found with greedy algorithm, where integer knapsack doesn't)
cf. Examples guarantees optimum : Seminar-room assignment
Seminar-room problem :

Get reservations from people who wants to use seminar room. People gives start time and finish time of their reservation.
Goal is to satisfy the most reservations.
We assign seminar-rooms in order of fastest finish time avoiding collision of previously reserved times.
Theorem :

Above algorithm guarantees optimal.
Proof.

Claim : There exists at least an optimal schedule that includes the meeting with the earliest finishing time.
Proof for claim :

Assume there exists an optimal schedule S not including the meeting, say m_1, with the earliest finishing time.
Let m_i be the meeting with the earliest finishing time in S.
Then, S-{m_i} U {m_1} is also an optimal schedule.
Therefore, selecting m_1 is safe in the sense that it does keep a path to an optimal schedule.
Thus, repeat the process with the set of meetings excluding m_1 and those conflicting to m_1.

Metroid :

It is related with independent set.
1. heredity property
2. exchange property
Def.

A metroid M = (S, I) of a finite set S and a non-empty set I consisting of "independent" subsets of S satisfying :

1. If B in I and A is subset of B, then A is in I. (heredity) (also empty set is in I)
2. If A in I, B in I and |A| < |B|, then there is some element x in B-A such that A U {x} is in I.(exchange)

An example :

Graphic metroid M_G = (E, F) where E is the set of edges, F is the set of acyclic subset of edges(which is same as set of trees = Forest).

Claim : M_G is metroid i.e. the set of forests is an independent set.

Proof for claim :

(1) A subset of forest is also a forest.(heredity)

(2) Consider two forests A and B s.t. |A| < |B|. Since A is a forest, A consists of zero or more (disconnected) trees.

For each tree of A of size k, B does not contain more than k edges connecting the vertices in the tree.

Thus, there is at least an edge e that connects two different trees of A.

A U {e} is also a forest. (e is extension of A)

Def. In a metroid M = (S, I) and A in I, if A U {x} is in I for x not in A, then we call x is an extension of A.

Def. In a metroid M = (S, I), A in I is called maximal if A does not have any extension.

Theorem :

In a metroid M = (S, I), every maximal subset of I has the same size.

Proof.

Assume for the contradiction there are two maximal subsets A, B in I of different sizes.

Let |A| < |B| without loss of generality.

By the property of exchange, ther is an element x in B-A such that A U {x} in I.

Contradiction to the fact that A is maximal.

Greedy algorithm on a weighted metroid :

On a metroid M = (S, I) where every element has a positive weight, we want to find a subset in I with the maximum weight sum.

cf. A subset with the maximum weight sum is always maximal.(Due to every weight is positive.)

```
Algorithm_Greedy(M, W) // M : metroid (S, I), W : weights
{
    A <- empty_set
    Sort S in decreasing order by W.
    for each x in S in decreasing order by W
        if (A U {x} in I) then
```

```
            A <- A U {x}
    return A
}
```

An example : kruskal algorithm for minimum spanning trees.

Theorem :

The algorithm greedy guarantees a mximum-weight independent set.

Proof.

Assume to the contrary that there is an independent subset X with a greater weight sum than that of the subset A, the output of greedy.
since A and X arae both maximal subset, |A| = |X|.
Let the weights of A be a_1 >= a_2 >= ... >= a_n and the weights of X be x_1 >= x_2 >= ... >= x_n.
(And let a_1, a_2, ..., a_n be the sequence added to A.)
Since W(X) > W(A), there is at least an x_i such that x_i > a_i.
Consider A_(i-1) = {a_1, a_2, ..., a_(i-1)} and X_i = {x_1, x_2, ..., x_i}
By the property of exchange, there is at least an x_k in {X_i - A_(i-1)}
such that A_(i-1) U {x_i} in I.
Since all x_k's are greater than a_i, the algorithm Greedy chooses the greatest x_k among them.
Therefore, if there exists a better subset then A, the algorithm Greedy never returns A.
this means that if the algorithm Greedy returns A, then there is no better solution.