Divide-and-Conquer: Searching in an Array:

Searching in an array

Input: An array A with n elements. A key k.

Output: An index i, where $A[i] = k$. If there is no such i, then NOT_FOUND.

```
LinearSearch(A, low, high, key):
if high < low:
    return NOT_FOUND
if A[low] = key:
    return low
return LinearSearch(A, low + 1, high, key)
```

Recurrence relation:

A recurrence relation is an equation recursively defining a sequence of values.

Example (Fibonacci recurrence relation):

$$F(n) = \begin{cases} 0 \text{ if } n = 0 \\ 1 \text{ if } n = 1 \\ F(n-1) + F(n-2) \text{ if } n > 1 \end{cases}$$

Recurrence defining worst-case time for linear search:

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

$T(0) = c$

So, the time complexity is $\Theta(n)$ because

$$T(n) = T(n-1) + c = T(n-2) + 2c = \ldots = cn = \Theta(n)$$

where c is a constant

```
LinearSearchIt(A, low, high, key):
for i from low to high:
    if A[i] = key:
        return i
return NOT_FOUND
```

Searching in a sorted array

Input: A sorted array $A[\text{low}\ldots\text{high}]$ $(\forall \text{low} \leq i < \text{high} : A[i] \leq A[i+1])$. A key k.

Output: An index, i, $(\text{low} \leq i < \text{high})$ where $A[i] = k$. Otherwise, the greatest index i where $A[i] < k$. Otherwise, $k < A[\text{low}]$, the results is $\text{low} - 1$.

```
BinarySearch(A, low, high, key):
if high < low:
    return low - 1
mid <- floor(low + (high - low)/2)
if key = A[mid]:
    return mid
else if key < A[mid]:
    return BinarySearch(A, low, mid - 1, key)
else:
    return BinarySearch(A, mid + 1, high, key)
```

$T(n) = T(\lfloor \frac{n}{2} \rfloor) + c$

$T(0) = c$

So, the time complexity is $\Theta(\log_2 n)$ because

$$T(n) = T(\frac{n}{2}) + c = T(\frac{n}{4}) + c + c = \sum_{i=0}^{\log_2 n} c = \Theta(\log_2 n)$$
$$\frac{n}{2^h} = 1 \text{ when } h = \log_2 n$$

Divide-and-Conquer: Polynomial multiplication

Multiplying polynomials

Input: Two $n-1$ degree polynomials: $a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x + a_0$
$b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \ldots + b_1 x + b_0$

Output: The product polynomial: $c_{2n-2}x^{2n-2} + c_{2n-3}x^{2n-3} + \ldots + c_1 x + c_0$ where
$c_{2n-2} = a_{n-1}b_{n-1}$
$c_{2n-3} = a_{n-1}b_{n-2} + a_{n-2}b_{n-1}$
$\ldots$
$c_2 = a_2 b_0 + a_1 b_1 + a_0 b_2$
$c_1 = a_1 b_0 + a_0 b_1$
$c_0 = a_0 b_0$

```
MultPoly(A, B, n):
pair <- Array[n][n]
for i from 0 to n - 1:
    for j from 0 to n - 1:
        pair[i][j] <- A[i]*B[j]
product <- Array[2n - 1]
for i from 0 to 2n - 1:
    product[i] <- 0
for i from 0 to n - 1:
    for j from 0 to n - 1:
        product[i + j] <- product[i + j] + pair[i][j]
return product
```

The naive solution requires `O(n^2)` because multiplication of n^2 pairs of coefficients.

Naive Divide-and-conquer method:

Let $A(x) = D_1(x)x^{\frac{n}{2}} + D_0(x)$ where
$D_1(x) = a_{n-1}x^{\frac{n}{2}-1} + a_{n-2}x^{\frac{n}{2}-2} + ... + a_1x + a_{\frac{n}{2}}$
$D_0(x) = a_{\frac{n}{2}-1}x^{n-1} + a_{\frac{n}{2}-2}x^{n-2} + ... + a_1x + a_0$

Let $B(x) = E_1(x)x^{\frac{n}{2}} + E_0(x)$ where $E_1(x) = b_{n-1}x^{\frac{n}{2}-1} + b_{n-2}x^{\frac{n}{2}-2} + ... + b_1x + b_{\frac{n}{2}}$
$E_0(x) = b_{\frac{n}{2}-1}x^{n-1} + b_{\frac{n}{2}-2}x^{n-2} + ... + b_1x + b_0$

Then,
$AB = (D_1(x)x^{\frac{n}{2}} + D_0)(E_1(x)x^{\frac{n}{2}} + E_0) = (D_1E_1)x^n + (D_1E_0 + D_0E_1)x^{\frac{n}{2}} + D_0E_0$

We just need to calculate $D_1E_1, D_1E_0, D_0E_1, D_0E_0$

Recurrence: $T(n) = 4T(\frac{n}{2}) + kn$

```
Mult2(A, B, n, a[l], b[l]):
R = array[0...2n - 2]
if n = 1:
    R[0] = A[a[l]]*B[b[l]]; return R
R[0...n - 2] = Mult2(A, B, n/2, a[l], b[l])
R[n...2n - 2] = Mult2(A, B, n/2, a[l] + n/2, b[l] + n/2)
D[0]E[1] = Mult2(A, B, n/2, a[l], b[l] + n/2)
D[1]E[0] = Mult2(A, B, n/2, a[l] + n/2, b[l])
R[n/2...n + n/2 - 2] += D[1]E[0] + D[0]E[1]
return R
```

So, the time complexity is $T(n) = 4T(\frac{n}{2}) + kn = \sum_{i=0}^{\log_2 n} 4^i k \frac{n}{2^i} = \Theta(n^2)$

Karatsuba approach:

$A(x) = a_1x + a_0$

$B(x) = b_1(x) + b_0$

$C(x) = a_1 b_1 x^2 + (a_1 b_0 + a_0 b_1)x + a_0 b_0$

Needs 4 multiplications

Rewrite as

$C(x) = a_1 b_1 x^2 + ((a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0)x + a_0 b_0$

Needs 3 multiplication.

So, the time complexity is
$T(n) = 3T(\frac{n}{2}) + kn = \sum_{i=0}^{\log_2 n} 3^i k \frac{n}{2^i} = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.58})$

Divide-and-Conquer: Master theorem

Theorem (Master theorem):

If $T(n) = aT(\lceil \frac{n}{b} \rceil) + O(n^d)$ (for constants $a > 0, b > 1, d \geq 0$), then:

$T(n) = \begin{cases} O(n^d) \text{ if } d > \log_b a \\ O(n^d \log n) \text{ if } d = log_b a \\ O(n^{\log_b a}) \text{ if } d < log_b a \end{cases}$

Proof:

Level	Cost
0	$O(n^d)$
1	$aO((\frac{n}{b})^d)$ $= \frac{a}{b^d} O(n^d)$
...	...
$\log_b n$	$a^{\log_b n} O((\frac{n}{b^{\log_b n}})^d) = a^{\log_b n} = O(n^{\log_b a})$

It is because

$$a^{\log_b n} = b^{\log_b a^{\log_b n}} = b^{\log_b n \log_b a} = b^{\log_b n^{\log_b a}} = n^{\log_b a}$$

So, total cost is

$$O(n^d) + \frac{a}{b^d} O(n^d) + \frac{a^2}{b^{2d}} O(n^d) + \ldots + O(n^{\log_b a}) = \sum_{i=0}^{\log_b n} O(n^d)(\frac{a}{b^d})^i$$

Case 1: $\frac{a}{b^d} < 1$ or $d > \log_b a$

$$\sum_{i=0}^{\log_b n} O(n^d)(\frac{a}{b^d})^i = O(n^d)$$

because the costs at level 1, ..., level n are less than a constant times $O(n^d)$. By definition of big-O, it is also $O(n^d)$.

Case 2: $\frac{a}{b^d} = 1$ or $d = \log_b a$

$$\sum_{i=0}^{\log_b n} O(n^d)(\frac{a}{b^d})^i = \sum_{i=0}^{\log_b n} O(n^d) = (1 + \log_b n)O(n^d) = \log_b nO(n^d) = O(n^d \log n)$$

Case 3: $\frac{a}{b^d} > 1$ or $d < \log_b a$

$$\sum_{i=0}^{\log_b n} O(n^d)(\frac{a}{b^d})^i = O(O(n^d)(\frac{a}{b^d})^{\log_b n}) = O(O(n^d)(\frac{a^{\log_b n}}{b^{d \log_b n}})) = O(O(n^d)(\frac{n^{\log_b a}}{n^d})) = O(n^{\log_b a})$$

Divide-and-Conquer: Sorting problem

Sorting

Input: Sequence $A[1...n]$

Output: Permutation $A'[1...n]$ of $A[1...n]$ in a non-decreasing order

```
SelectionSort(A[1...n]):
for i from 1 to n:
    minIndex <- i
    for j from i + 1 to n:
        if A[j] < A[minIndex]:
            minIndex <- j
    swap(A[i], A[minIndex])
```

The running time of `SelectionSort(A[1...n])` is $O(n^2)$ because n iterations of outer for-loop and at most $n - 1$ iterations of inner for-loop. It is a sort in place which requires a constant amount of extra memory.

```
MergeSort(A[1...n]):
if n = 1:
    return A
m <- floor(n/2)
B <- MergeSort(A[1...m])
C <- MergeSort(A[m + 1...n])
A' <- Merge(B, C)
return A'

Merge(B[1...p], C[1...q]):
# B and C are sorted
D <- empty array of size p + q
```

```
while B and C are both non-empty:
    b <- the first element of B
    c <- the first element of C
    if b <= c:
        move b from B to the end of D
    else:
        move c from C to the end of D
move the rest of B and C to the end of D
# One of B and C must be empty
return D
```

The running time of `MergeSort(A[1...n])` is $O(n \log n)$. The running time of merging B and C is $O(n)$. Hence, the running time of `MergeSort(A[1...n])` satisfies a recurrence $T(n) \leq 2T(\frac{n}{2}) + O(n)$.

Comparison based sorting algorithm

A comparison based sorting algorithm sorts objects by comparing pairs of them. For example, selection sort and merge sort.

Lemma:

Any comparison based sorting algorithm performs $\Omega(n \log n)$ comparisons in the worst case to sort n objects.

We can represent the comparison by a decision tree with $n!$ leaves. Let the depth of the tree be d and it satisfies $d \geq \log_2 (\text{total number of leaves})$. So, the running time is at least $\log_2(n!) = \Omega(n \log n)$.

Proof:

$$\log_2(n!) = \log_2(1 \cdot 2 \cdot \ldots \cdot n) = \log_2 1 + \log_2 2 + \ldots + \log_2 n \geq \log_2 \frac{n}{2} + \ldots + \log_2 n \geq \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n)$$

No comparison based sorting algorithm can do better asymptotically.

Non-comparison based sorting algorithm

```
CountSort(A[1...n]):
Count[1...M] <- [0,...,0]
for i from 1 to n:
    Count[A[i]] <- Count[A[i]] + 1 # A[i] contains integer, Count[A[i]]
stores the total count
Pos[1...M] <- [0,...,0]
Pos[1] <- 1
for j from 2 to M:
    Pos[j] <- Pos[j - 1] + Count[j - 1] # Beginning position of each
integer
for i from 1 to n:
    A'[Pos[A[i]]] <- A[i]
    Pos[A[i]] <- Pos[A[i]] + 1 # Update the position of each integer for
the next iteraion
```

Lemma:

Provided that all elements of `A[1...n]` are integers from 1 to M, `CountSort(A)` sorts `A` in time `O(n + m)`.

Divide-and-Conquer: Quick Sort

```
QuickSort(A, l, r):
if l >= r:
    return
m <- Partition(A, l, r)
QuickSort(A, l, m - 1)
QuickSort(A, m + 1, r)

Partition(A, l, r):
x <- A[l] # pivot, at the beginning, take l = 0 and r = length(A)
j <- l
for i from l + 1 to r:
    if A[i] <= x:
        j <- j + 1 # Swap with itself if there is no A[i] > x
        swap A[j] and A[i] # A[l + 1...j] <= x and A[j + 1,...,i] > x
swap A[l] and A[j] # At last, A[j] <= x at pos j where j > l. So, swap
A[l] and A[j]
return j
```

Randomized Quick Sort

Unbalanced partition can leads to `O(n^2)` running time. Suppose the minimum element is selected as an pivot. The array will be partitioned into two subarrays, with the first one having 1 element (the pivot) and the second one having $n - 1$ elements. The `Partition(A, l, r)` takes `O(n)` comparisons.

$$T(n) = n + T(n-1) = n + (n-1) + (n-2) + \ldots + 1 = \Theta(n^2)$$

For balanced partition, if the pivot partitions the array into two equal-sized subarrays, we have the following recurrence relation:

$$T(n) = 2T(\frac{n}{2}) + n = \Theta(n \log n)$$

```
RandomizedQuickSort(A, l, r):
if l >= r:
    return
k <- random number between l and r
swap A[l] and A[k]
m <- Partition(A, l, r)
RandomizedQuickSort(A, l, m - 1)
RandomizedQuickSort(A, m + 1, r)
```

Theorem

Assume that all the elements of `A[1...n]` are pairwise different. Then the average running time of `RandomizedQuickSort(A)` is `O(nlogn)` while the worst case running time is `O(n^2)`.

Remark: Averaging is over random numbers used by the algorithm but not over the inputs.

Proof:

$$\text{For } i < j, \chi_{ij} = \begin{cases} 1, \text{A'[i] and A'[j] are compared} \\ 0, \text{otherwise} \end{cases}$$

For $i < j$, either $A'[i]$ and $A'[j]$ are compared exactly once or not compared at all as we compare with a pivot. It implies the worst case running time is $O(n^2)$.

Crucial observation: $\chi_{ij} = 1$ iff the first selected pivot in $A'[i, \ldots, j]$ is $A'[i]$ or $A'[j]$. Then, $P(\chi_{ij}) = \frac{2}{j-i+1}$ and $E(\chi_{ij}) = \frac{2}{j-i+1}$ as χ_{ij} is Bernoulli-distributed. Then, the expected value of the running time is

$$E[\sum_{i=1}^{n}\sum_{j=i+1}^{n} \chi_{ij}] = \sum_{i=1}^{n}\sum_{j=i+1}^{n} E[\chi_{ij}] = \sum_{i<j} \frac{2}{j-i+1} \leq 2n(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}) = \Theta(n \log n)$$

Remark: If i is fixed, then

$$\sum_{i<j} \frac{2}{j-i+1} = 2\sum_{i=1}^{n}(\frac{1}{3-i} + \frac{1}{4-i} + \ldots + \frac{1}{n+1-i}) \leq 2\sum_{i=1}^{n}(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}) = 2n(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n})$$

Remark: If two elements are far from each other in the sorted array, then the probability of being compared is low because there exist many other pivots to make comparison indirectly. If two elements are next to each other in the sorted array, then the probability of being compared is 1 because there is no pivot in between two elements to separate them into two different subarrays in quick sort algorithm.

Equal element problem

```
RandomizedQuickSort(A, l, r):
if l >= r:
    return
k <- random number between l and r
swap A[l] and A[k]
(m[1], m[2]) <- Partition3(A, l, r)
RandomizedQuickSort(A, l, m[1] - 1)
RandomizedQuickSort(A, m[2] + 1, r)
```

`Partition3(A, l, r)` does the follows:

$$\forall l \le k \le m_1 - 1, A[k] < x$$
$$\forall m_1 \le k \le m_2, A[k] = x$$
$$\forall m_2 + 1 \le k \le r, A[k] > x$$

Tail Recursion Elimination

```
QuickSort(A, l, r):
while l < r: # We realize that we can eliminate the last recursive call
    m <- Partition(A, l, r)
    QuickSort(A, l, m - 1)
    l <- m + 1

QuickSort(A, l, r):
while l < r:
    m <- Partition(A, l, r)
    if (m - l) < (r - m): # Two recursive calls are independent, there
is only one call
        QuickSort(A, l, m - 1)
        l <- m + 1
    else:
        QuickSort(A, m + 1, r)
        r <- m - 1
```

Worst-case space requirement: `O(logn)`