Analysis of Algorithms: Divide and Conquer Notes

Divide And Conquer

- A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly.

Insertion Sort

- ***Algorithm:***
    - Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list.
    - Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.
    - Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.
- ***Pseudo Code:***

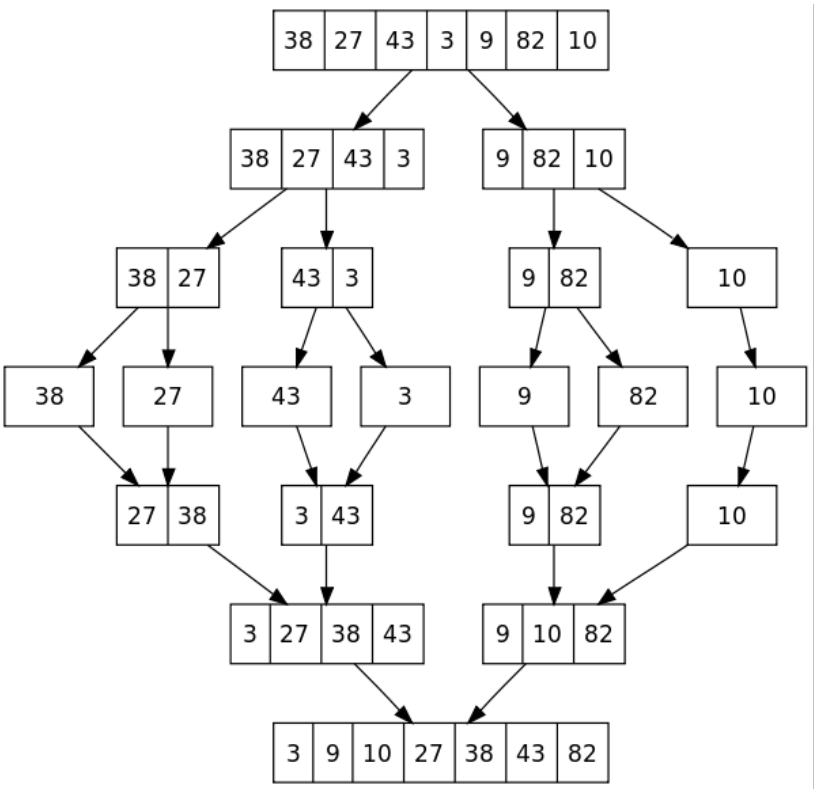
```
for i = 1 to length(A) – 1
    x = A[i]
    j = i – 1
    while j >= 0 and A[j] > x
        A[j+1] = A[j]
        j = j – 1
    end while
```

```
            A[j+1] = x
        end for
```

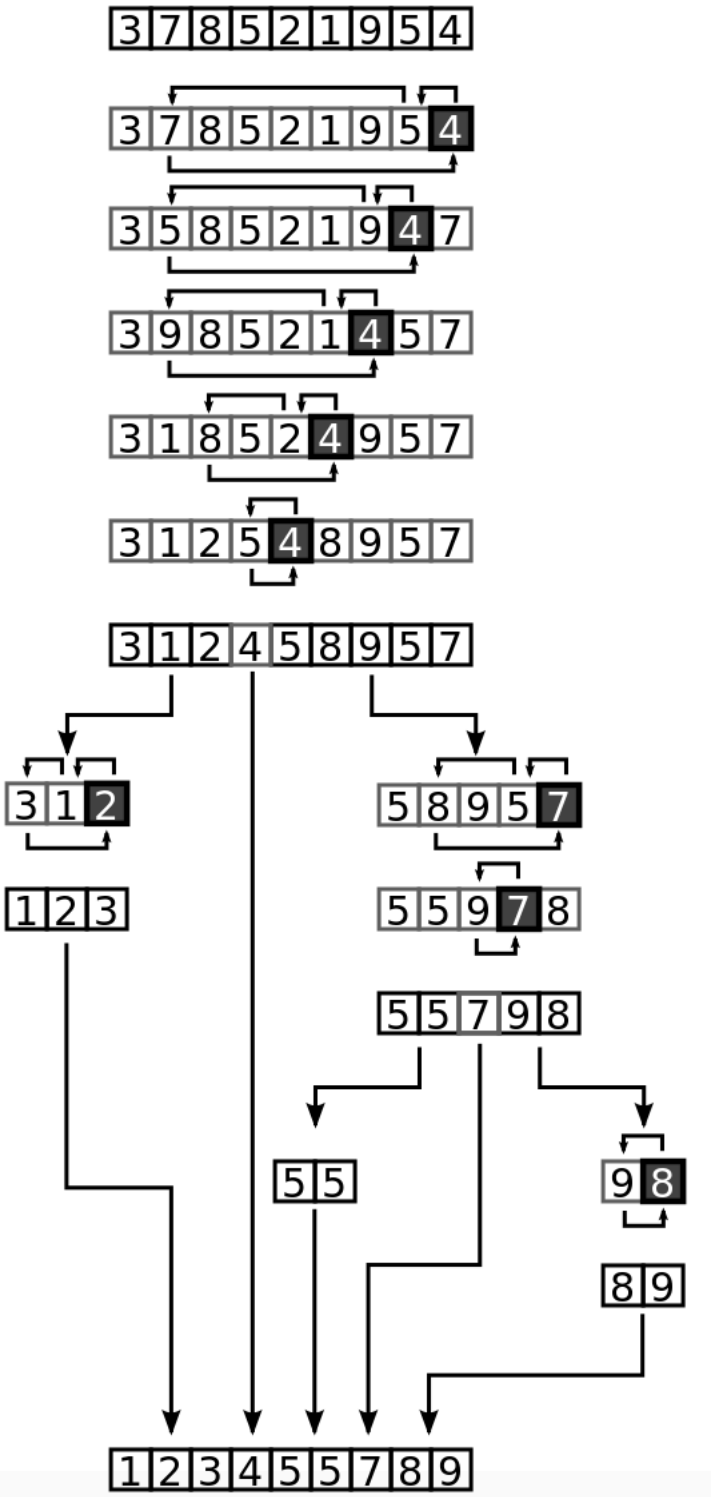
- ***Example:***
    - The following table shows the steps for sorting the sequence {3, 7, 4, 9, 5, 2, 6, 1}. In each step, the item under consideration is underlined. The item that was moved (or left in place because it was biggest yet considered) in the previous step is shown in bold.
    - 3 7 4 9 5 2 6 1
      3 7 4 9 5 2 6 1
      3 **7** 4 9 5 2 6 1
      3 **4** 7 9 5 2 6 1
      3 4 7 **9** 5 2 6 1
      3 4 **5** 7 9 2 6 1
      2 3 4 5 7 9 6 1
      2 3 4 5 **6** 7 9 1
      1 2 3 4 5 6 7 9
- ***Analysis of Insertion Sort:***
    - Suppose that upon every call to insert, the value being inserted is less than every element in the subarray to its left. When we call insert the first time, k=1. The second time, k=2. The third time, k=3. And so on, up through the last time, when k = n-1. Therefore, the total time spent inserting into sorted subarrays is:
      $$c \cdot 1 + c \cdot 2 + c \cdot 3 + \cdots c \cdot (n-1) = c \cdot (1 + 2 + 3 + \cdots + (n-1))$$
    - That sum is an arithmetic series, except that it goes up to n-1 rather than n. Using our formula for arithmetic series, we get that the total time spent inserting into sorted subarrays is:
      $$c \cdot (n-1+1)((n-1)/2) = cn^2/2 - cn/2$$
    - Using big-Θ notation, we discard the low-order term cn/2 and the constant factors c and 1/2, getting the result that the running time of insertion sort, in this case, is $O(n^2)$.

- Worst Case Time Complexity: $O(n^2)$
  Best Case Time Complexity: $O(n)$
  Average Case Time Complexity: $O(n^2)$
- https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort

Merge Sort

- ***Algorithm:***
  - Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
  - Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.
  - (1) Divide the list in half.
    (2) Merge sort the first half.
    (3) Merge sort the second half.
    (4) Merge both halves back together.
- ***Pseudo Code:***
  - ```
    MergeSort (Array(First…Last))
    Begin
    If Array contains only one element then:
        Return Array
    Else
        Middle = ((Last + First)/2) rounded down to the
        nearest integer
        LeftHalfArray = MergeSort(Array(First…Middle))
        RightHalfArray = MergeSort(Array(Middle+1…Last))
        ResultArray = Merge(LeftHalfArray, RightHalfArray)
        Return ResultArray
    EndIf
    End MergeSort
    ```
- ***Example:***
    ```

- A recursive merge sort algorithm used to sort an array of 7 integer values. Top-Down.



- ***Analysis of Merge Sort:***
  - The base case occurs when n = 1.
  - When n >= 2, time for merge sort steps:
    1) Divide: Just compute q as the average of p and r, which takes constant time O(1).
    2) Conquer: Recursively solve 2 subproblems, each of size n/2.
    3) Combine: Merge on an n-element subarray takes O(n) time.

    $T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases}$
  - Using Master's Theorem: T(n) = O(n log n)
  - Worst Case Time Complexity: O(n log n)
    Best Case Time Complexity: O(n log n)
    Average Case Time Complexity: O(n log n)

- https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort

Quick Sort

- ***Algorithm:***
  - 1) Pick an element, called a pivot, from the list.
    2) Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
    3) Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.
- ***Pseudo Code:***
  - ```
    algorithm quicksort(A, lo, hi) is
        if lo < hi then
            p := partition(A, lo, hi)
            quicksort(A, lo, p-1)
            quicksort(A, p+1, hi)

    algorithm partition(A, lo, hi) is
        pivot := A[hi]
        i := lo    // place for swapping
        for j := lo to hi-1 do
            if A[j] <= pivot then
               swap A[i] with A[j]
               i := i + 1
        swap A[i] with A[hi]
        return i
    ```
- ***Example:***
  - Full example of quicksort on a random set of numbers. The shaded element is the pivot. It is always chosen as the last element of the partition.
    ```



- ***Analysis of Quick Sort:***

- **WORST CASE:** When quicksort always has the most unbalanced partitions possible, then the original call takes cn time for some constant c*c*, the recursive call on n-1 elements takes c(n-1) time, the recursive call on n-2 elements takes c(n-2) time, and so on.

$$\begin{aligned} cn + c(n-1) + c(n-2) + \cdots + 2c &= c(n + (n-1) + (n-2) + \cdots + 2) \\ &= c((n+1)(n/2) - 1)\ . \end{aligned}$$

  = $O(n^2)$

- **BEST CASE:** Quicksort's best case occurs when the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other. The former case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has (n-1)/2 elements. **=** O(n log n)

- Worst Case Time Complexity: $O(n^2)$
  Best Case Time Complexity: O(n log n)
  Average Case Time Complexity: O(n log n)

- https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Quick Selection

- ***Algorithm:***
  - We can modify quicksort to perform selection for us. Observe that when we choose a pivot v and use it to partition the list into three lists I1, Iv, and I2, we know which of the three lists contains index j, because we know the lengths of I1 and I2. Therefore, we only need to search one of the three lists.
  - Here's the quickselect algorithm for finding the item at index j - that is, having the (j + 1)th smallest key.
  - Start with an unsorted list I of n input items.

- Choose a pivot item v from I.
- Partition I into three unsorted lists I1, Iv, and I2.
  - I1 contains all items whose keys are smaller than v's key.
  - I2 contains all items whose keys are larger than v's.
  - Iv contains the pivot v.
  - Items with the same key as v can go into any of the three lists.

  (In list-based quickselect, they go into Iv; in array-based quickselect, they go into I1 and I2, just like in array-based quicksort.)
-
```
if (j < |I1|) {
    Recursively find the item with index j in I1;
    return it.
}
else if (j < |I1| + |Iv|) {
     Return the pivot v.
}
else {   // j >= |I1| + |Iv|.
     Recursively find the item with index j - |I1| -
     |Iv| in I2; return it.
}
```

- ***Pseudo Code:***
  -
- ***Example:***
  -
- ***Analysis of Quick Selection:***
  - O(n)

Selection Sort

- ***Algorithm:***
  - The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the

front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list.

- Initially, the sorted sublist is empty and the unsorted sublist is the entire input list.
- The algorithm proceeds by finding the smallest (or largest, depending on the sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

- ***Pseudo Code:***

```
/* a[0] to a[n-1] is the array to sort */
int i,j;
/* advance the position through the entire array */
/* (could do j < n-1 because single element is also
min element) */
for (j = 0; j < n-1; j++) {
    /* find the min element in the unsorted a[j .. n-
      1] */
    /* assume the min is the first element */
    int iMin = j;
    /* test against elements after j to find the
    smallest */
    for ( i = j+1; i < n; i++) {
        /* if this element is less, then it is the new
        minimum */
        if (a[i] < a[iMin]) {
          /* found new minimum; remember its index */
          iMin = i;
        }
    }
    if(iMin != j) {
        swap(a[j], a[iMin]);
    }
}
```

- ***Example:***
  - 64 25 12 22 11 // this is the initial, starting state of the array
    11 25 12 22 64 // sorted sublist = {11}

11 12 25 22 64 // sorted sublist = {11, 12}
11 12 22 25 64 // sorted sublist = {11, 12, 22}
11 12 22 25 64 // sorted sublist = {11, 12, 22, 25}
11 12 22 25 64 // sorted sublist = {11, 12, 22, 25, 64}

- ***Analysis of Selection Sort:***
    - Adding up the running times for the three parts, we have $O(n^2)$ for the calls to indexOfMinimum, O(n) for the calls to swap, and O(n) for the rest of the loop in selectionSort. The $O(n^2)$ term is the most significant, and so we say that the running time of selection sort is $O(n^2)$.
    - Notice also that no case is particularly good or particularly bad for selection sort. The loop in indexOfMinimum will always make n^2+ n/2*n* iterations, regardless of the input. Therefore, we can say that selection sort runs in $O(n^2)$ time in all cases.
    - Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array.
    - Worst Case Time Complexity: $O(n^2)$
      Best Case Time Complexity: $O(n^2)$
      Average Case Time Complexity: $O(n^2)$
    - https://www.khanacademy.org/computing/computer-science/algorithms/sorting-algorithms/a/analysis-of-selection-sort

Bubble Sort

- ***Algorithm:***
    - Repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order.

- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

- ***Pseudo Code:***

```
procedure bubbleSort(A : list of sortable items)
    n = length(A)
    repeat
      swapped = false
      for i = 1 to n-1 inclusive do
        /* if this pair is out of order */
        if A[i-1] > A[i] then
          /*swap them and remember something changed*/
          swap(A[i-1],A[i])
          swapped = true
        end if
      end for
    until not swapped
end procedure
```

- ***Example:***

  - Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.

  - First Pass
    (**5 1** 4 2 8) → (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
    (1 **5 4** 2 8) → (1 **4 5** 2 8), Swap since 5 > 4
    (1 4 **5 2** 8) → (1 4 **2 5** 8), Swap since 5 > 2
    (1 4 2 **5 8**) → (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

  - Second Pass
    (**1 4** 2 5 8) → (**1 4** 2 5 8)
    (1 **4 2** 5 8) → (1 **2 4** 5 8), Swap since 4 > 2
    (1 2 **4 5** 8) → (1 2 **4 5** 8)
    (1 2 4 **5 8**) → (1 2 4 **5 8**)

Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

- Third Pass
  (**1 2** 4 5 8) → (**1 2** 4 5 8)
  (1 **2 4** 5 8) → (1 **2 4** 5 8)
  (1 2 **4 5** 8) → (1 2 **4 5** 8)
  (1 2 4 **5 8**) → (1 2 4 **5 8**)

- ***Analysis of Bubble Sort:***
  - For an array of size n, in the worst case: 1st passage through the inner loop: n – 1 comparisons and n – 1 swaps.
  - (n-1)st passage through the inner loop: one comparison and one swap.
  - All together: c((n-1) + (n-2) + … + 1), where c is the time required to do one comparison, one swap, check the inner loop condition and increment.
  - Equation: c * n(n-1)/2 + k + c1(n-1) = $O(n^2)$
  - Worst Case Time Complexity: $O(n^2)$
    Best Case Time Complexity: O(n)
    Average Case Time Complexity: $O(n^2)$

Bubble Sort

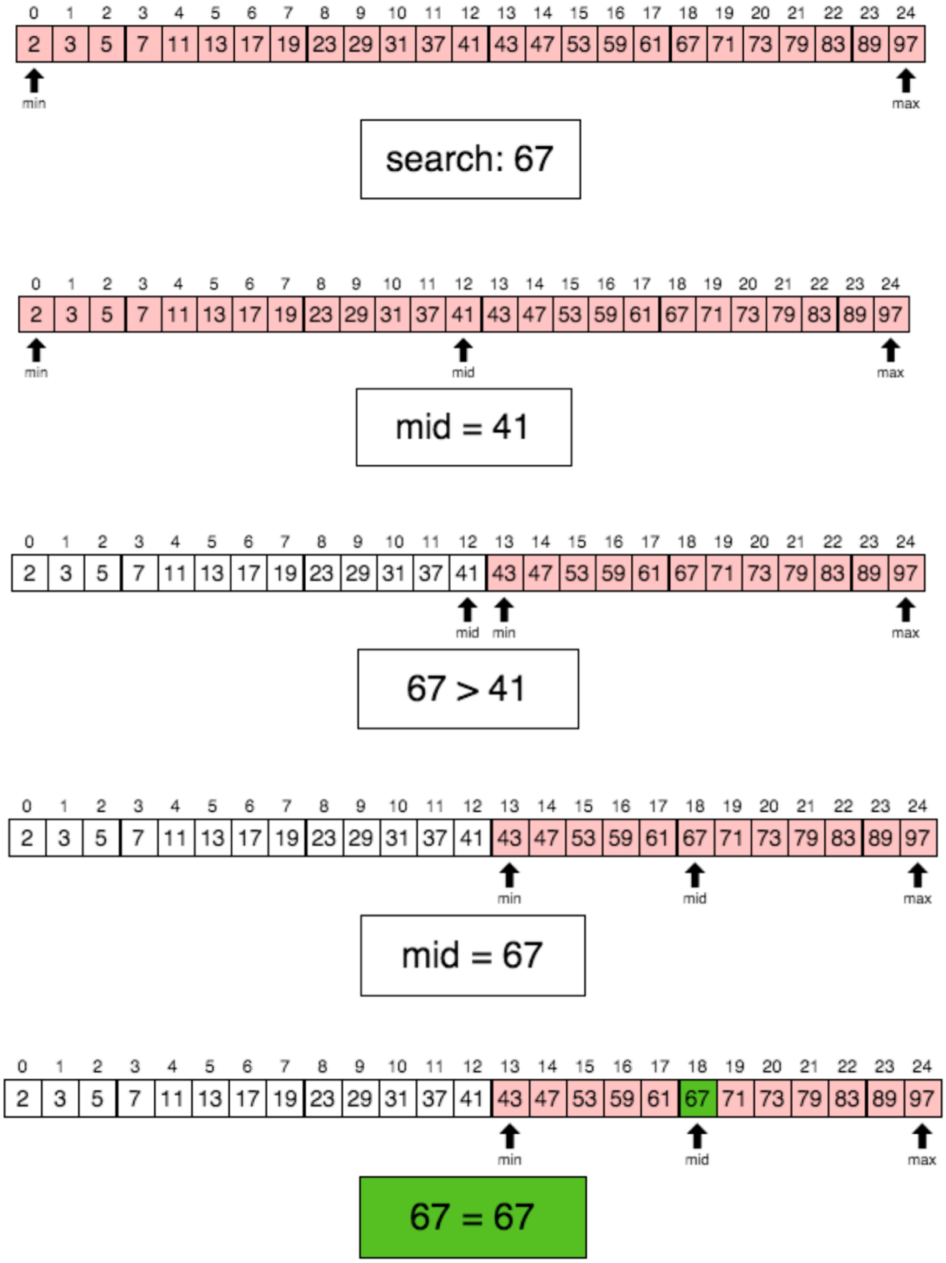
- ***Algorithm:***
  - The binary search algorithm begins by comparing the target value to the value of the middle element of the sorted array.
  - If the target value is equal to the middle element's value, then the position is returned and the search is finished.

- If the target value is less than the middle element's value, then the search continues on the lower half of the array; or if the target value is greater than the middle element's value, then the search continues on the upper half of the array.
- This process continues, eliminating half of the elements, and comparing the target value to the value of the middle element of the remaining elements - until the target value is either found (and its associated element position is returned), or until the entire array has been searched (and “not found” is returned).

- ***Pseudo Code:***

```
int binary_search(int A[], int key, int imin, int imax)
{
  // test if array is empty
  if (imax < imin)
    // set is empty, so return value showing not found
    return KEY_NOT_FOUND;
  else
    {
      // calculate midpoint to cut set in half
      int imid = midpoint(imin, imax);

      // three-way comparison
      if (A[imid] > key)
        // key is in lower subset
        return binary_search(A, key, imin, imid - 1);
      else if (A[imid] < key)
        // key is in upper subset
        return binary_search(A, key, imid + 1, imax);
      else
        // key has been found
        return imid;
    }
}
```

- ***Example:***



- ***Analysis of Binary Search:***
    - Worst Case Time Complexity: O(log n)
      Best Case Time Complexity: O(1)
      Average Case Time Complexity: O(log n)

Master's Theorem

Theorem 5.1 *Let a be an integer greater than or equal to 1 and b be a real number greater than 1. Let c be a positive real number and d a nonnegative real number. Given a recurrence of the form*

$$T(n) = \begin{cases} aT(n/b) + n^c & \text{if } n > 1 \\ d & \text{if } n = 1 \end{cases}$$

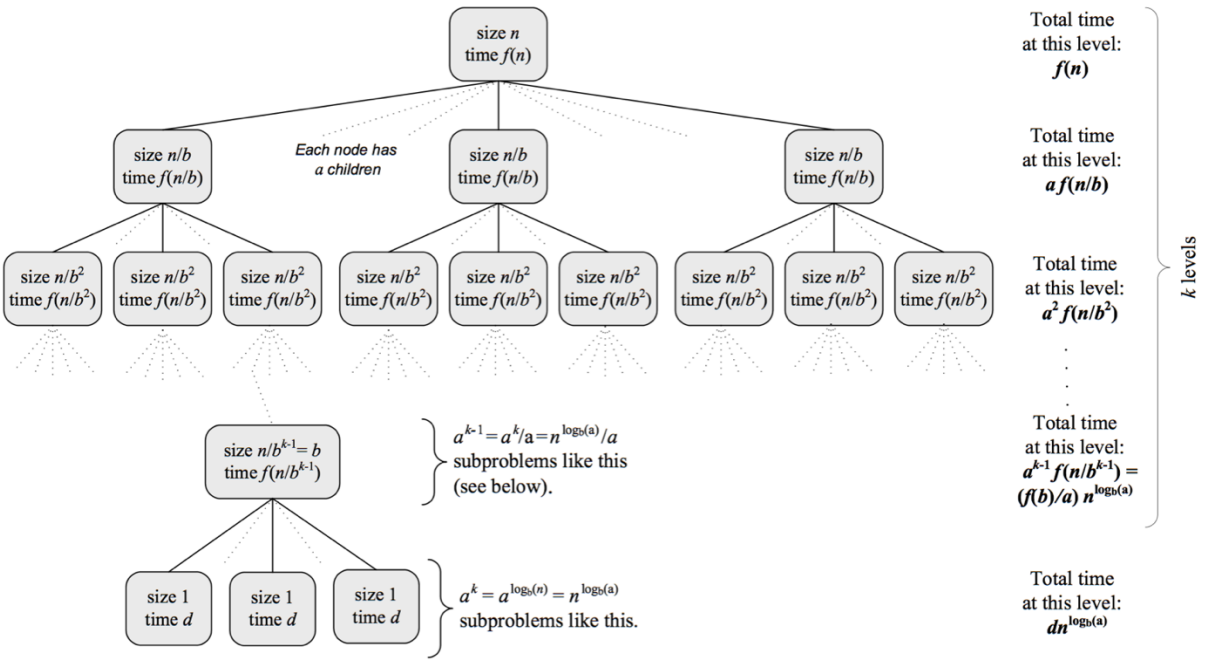
then for n a power of b,

1. *if* $\log_b a < c$, $T(n) = \Theta(n^c)$,
2. *if* $\log_b a = c$, $T(n) = \Theta(n^c \log n)$,
3. *if* $\log_b a > c$, $T(n) = \Theta(n^{\log_b a})$.

Recursion Tree

$$T(n) = f(n) + af(n/b) + \cdots + a^i f(n/b^i) + \cdots + a^L f(n/b^L)$$

The Recurrence $T(n) = aT(n/b) + f(n)$, $T(1) = d$, when n is a power of b ($n = b^k$, $k = \log_b(n)$)



$$T(n) = \text{total time} = f(n) + af(n/b) + a^2 f(n/b^2) + \ldots + a^{k-1} f(n/b^{k-1}) + dn^{\log_b(a)}$$

More generally, let us consider the recurrence:

$$T(n) = aT(n/b) + f(n).$$

Let us construct the recurrence tree.

FIGURE.

The depth of the recurrence tree is $L = \log_b n$. Thus,

$$T(n) = f(n) + af(n/b) + \cdots + a^i f(n/b^i) + \cdots + a^L f(n/b^L).$$

Log Rules

Rule 1: $\log_b (M \cdot N) = \log_b M + \log_b N$

Rule 2: $\log_b \left(\frac{M}{N}\right) = \log_b M - \log_b N$

Rule 3: $\log_b \left(M^k\right) = k \cdot \log_b M$

Rule 4: $\log_b (1) = 0$

Rule 5: $\log_b (b) = 1$

Rule 6: $\log_b \left(b^k\right) = k$

Rule 7: $b^{\log_b (k)} = k$

Where : $b > 1$, and M, N and k can be any real numbers

but M and N must be positive!

$$\log_b x = \frac{\log_a x}{\log_a b}$$

Finding Closest Pair Of Points

- Can be solved in $O(n^2)$ time by calculating distances of every pair of points and comparing the distances to find the minimum.
- Can calculate the smallest distance in O(nLogn) time using Divide and Conquer strategy.
- ***Algorithm:***
  1) Sort points according to their x-coordinates.
  2) Split the set of points into two equal-sized subsets by a vertical line $x = x_{mid}$.
  3) Solve the problem recursively in the left and right subsets. This yields the left-side and right-side minimum distances d_{Lmin} and d_{Rmin}, respectively.
  4) Find the minimal distance of d_{LRmin} among the set of pairs of points in which one point lies on the left of the dividing vertical and the second point lies to the right.
  5) The final answer is the minimum among d_{Lmin}, d_{Rmin}, and d_{LRmin}.

Strassen's Matrix Multiplication

- Given two square matrices A and B of size n x n each, find their multiplication matrix.
- ***Naïve Method:***
    - Time Complexity: $O(n^3)$

```
void multiply(int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

- ***Divide and Conquer:***

1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
2) Calculate the following values recursively. ae + bg, af + bh, ce + dg, and cf + dh.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

- Time Complexity: $O(n^3)$

- ***Strassen's Method:***

- The four sub-matrices of result are calculated using the following formulae:

p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

- Time Complexity: $O(N^{2.9074})$

Maximum Subsequence

- ***Algorithm 1:***
    - Try every slice $O(n^3)$
    1) Set maxSum = 0
    2) Calculate s(i,j) = sumOf a[k], and update the MAXSUM by s(i,j) if s(i,j) > MAXSUM ; for 1 < i < j < n
    3) MAXSUM is the answer
- ***Algorithm 2:***
    - Given sum from i to j – 1, we can compute the sum from i to j in constant time. This eliminates one nested loop, and reduces the running time to $O(n^2)$.
- ***Algorithm 3:***
    - Split the input sequence at midpoint. The max subsequence is entirely in the left half, entirely in the right half.
    - Max subsequences in each half found by recursion.
    - T = 1
      T(N) = 2T(N/2) + O(N)
      T(N) = O(N log N)
- ***Algorithm 4:***
    - Return the overall sum together with the max prefix and suffix sums (max subsequence sum, max prefix sum, max suffix sum, overall sum)
    - T(N) = T(N/2) + O(log N)
      T(N) = O(N)
    - Kadane's algorithm consists of a scan through the array values, computing at each position the maximum (positive sum)

subarray ending at that position. This subarray is either empty (in which case its sum is zero) or consists of one more element than the maximum subarray ending at the previous position.

Arithmetic and Geometric Series

Summation	Expansion	Convergence	Comments
$\sum_{n=0}^{n-1} r^n$	$= 1 + r + r^2 + r^3 + .. + r^{n-1}$ (first n terms)	for $r \neq 1$, $= (1 - r^n) / (1 - r)$ for $r = 1$, $= nr$	Finite Geometric Series
$\sum_{n=0}^{\infty} r^n$	$= 1 + r + r^2 + r^3 + ...$	for $\|r\| < 1$, converges to $1 / (1 - r)$ for $\|r\| >= 1$ diverges	Infinite Geometric Series

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

References:

https://www.khanacademy.org/computing/computer-science/algorithms

http://www.geeksforgeeks.org/fundamentals-of-algorithms/#DivideandConquer