I : Foundations

1 The Role of Algorithms in computing

1.1 Algorithms

Exercises

1.1-1: Give a real-world example that requires sorting or a real-world examples that require computing a convex hull.

→ Sorting: browse the price of the restaurants with ascending prices on NTU street, sorting of books,

→ Convex hull: Computing the diameter of set of points, pattern recognition, image processing, statistics, computer vision etc.

1.1-2: Other than Speed, what other measure of efficiency might one use in a real-world setting?

→ Memory efficiency & coding efficiency → Degree of parallelism of resources used, Accessibility

1.1-3: Select a data structure that you have seen previously & discuss its strength & its limitations.

→ Linked list:
- → Strengths: insertion & deletion
- → Limitations: random access

1.1-4: How are the shortest-path & traveling-salesman problems (→ P-times, → NP-complete) given above similar? How are they different?

→ similar: finding path with shortest distance. E.g. lowest overall distance

→ Different: traveling-salesman has more constraints.

1.1-5: Come up with a real-world problem in which only the best solution will do. Then come up with one in which a sol? that is approx. best is good enough.

→ Best: find the GCD of 2 positive integers → Approximate: Find the sol? of Diff. equation

1.2 : Algorithms as a technology

1.2-1 : Give an example of an application that requires algorithmic content at the application level, & discuss the function of the algorithms involved.

→ Drive navigation → fingerprint Matching → Aktinator etc.

1.2-2 : Suppose we are comparing implementations of insertion sort & merge sort on the same machine. For I/p of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n\log n$ steps. For which values of n does insertion sort beat merge sort?

$8n^2 < 64n\log n \Rightarrow \boxed{n < 8\log n}$

Now, try values of $n = 2^n$ then $2^n < 8\log 2^n \Rightarrow \boxed{2^n < n^8}$

$n=1$; $1 < 8\log 1 = 8*0 = 0$ (False)

$n=2$: $2 < 8\log 2 \approx 8 * 0.693 \approx 5$ (True)

$n=50$; $50 < 8\log 50 \approx 8* 3.912 \approx 31$ (False)

Hence, $\boxed{2 < n < 43}$ Ans.

1.2-3 : What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algo. whose running time is 2^n on the same machine?

$100n^2 < 2^n$

For $n=10$: $100(10)^2 < 2^{10} \Rightarrow 10,000 \not< 1024$ (False)

For $n=14$; $100(14)^2 < 2^{14} \Rightarrow 19,600 \not< 16,384$ [False]

For $n=15$; $100(15)^2 < 2^{15} \Rightarrow$ True.

Hence, $\boxed{n \geq 15}$ Ans

2 : Getting Started

2.1 : Insertion Sort :

→ Good for sorting the small number of elements.
→ like playing cards.
→ It is in-place algo.

```
INSERTION-SORT(A)                                   Cost    times
① for j = 2 to n                                    c1      n
② do key = A[j]                                     c2      n-1
③   // Insert A[j] into the sorted sequence A[1..j-1]  0      n-1
④   i = j-1                                         c4      n-1
⑤   while i > 0 & A[i] > key                        c5      Σ_{j=2}^{n} tj
⑥     do A[i+1] = A[i]                              c6      Σ_{j=2}^{n} (tj-1)
⑦       i = i-1                                     c7      Σ_{j=2}^{n} (tj-1)
⑧   A[i+1] = key.                                   c8      n-1
```

The running time of the algorithm is

$\sum_{\text{all stmt}}$ (Cost of stmt.) · (number of times stmt. is executed).

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j-1) + c_7\sum_{j=2}^{n}(t_j-1) + c_8(n-1)$$

Best Case : The array is already sorted

→ Always find that $A[i] \leq key$ upon the first time while loop test is run (when $i = j-1$).
→ All t_j are 1.

→ The running time is

$$T(n) = c_1(n) + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

→ Can express $T(n)$ as $an+b$ for constants a & b (that depends on the stmt costs c_i) ⇒ $T(n)$ is a linear function of n.

Worst Case: The array is in reverse sorted order.

→ Always find that $A[i] > key$ in while loop test.

→ Have to compare key with all elements to the left of the jth position ⇒ compare with $j-1$ elements.

→ Since the while loop exits because i reaches 0, there's one additional test after the $j-1$ tests ⇒ $t_j = j$.

i.e. $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ & $\sum_{j=2}^{n} (t_j - 1) = \sum_{j=1}^{n} (j-1)$

→ $\sum_{j=2}^{n} j$ is k/a arithmetic series & it is equal to $\frac{n(n+1)}{2}$

→ Since $\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\frac{n(n+1)}{2} - 1$.

→ Letting $k = j-1$, we see that $\sum_{j=2}^{n} (j-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$

→ Running time is

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

⇒ $T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$.

→ Can express $T(n)$ as $an^2 + bn + c$ for constant a, b, c (that again depends on stmt costs) ⇒ $T(n)$ is quadratic function of n.

Worst case & average-case analysis

We usually concentrate on finding the worst-case running time: the longest running time for any I/p of size n.

Reasons:

→ The worst-case running time gives a guaranteed upper bound on the running time for any I/p.

→ For some algorithms, the worst case occurs often, Ex:- when searching for ~~time for a~~, the worst case often occurs when the item being searched for is not present, & searches for absent items may be frequent.

→ Why not analyze the average case? Because it is often about as bad as the worst case.

Ex:- Suppose that we randomly choose n numbers as the I/p to insertion sort.

→ On Average, the key in $A[j]$ is less than half the elements in $A[1 \ldots j-1]$ & it's greater than the other half.

⇒ On avg., the while loop has to look halfway through the

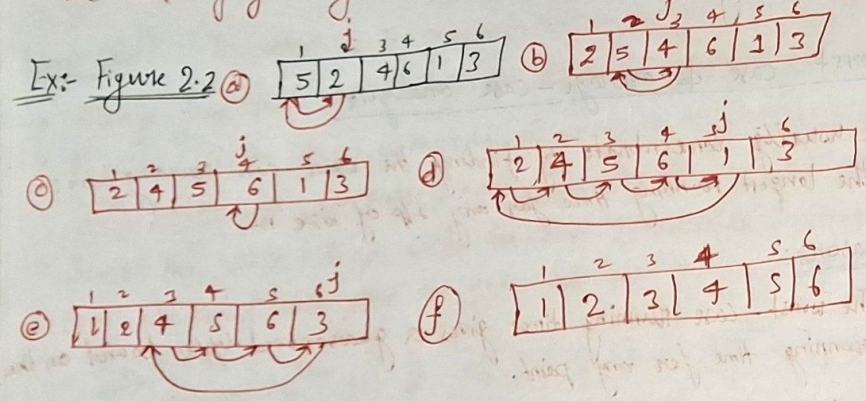
the sorted subarray $A[1 \ldots j-1]$ to decide where to drop key.

$\Rightarrow t_j = j/2$

Although the avg.-case running time is approximately half of the worst-case running time, it's still a quadratic function of n.

We say that the running time is $\Theta(n^2)$ to capture the notion that the order of growth is n^2.

Ex:- Figure 2.2



Correctness:

Loop invariant: At the start of each iteration of the "outer" for loop — the loop indexed by j — the subarray $A[1 \ldots j-1]$ consists of the elements originally in $A[1 \ldots j-1]$ but in sorted order.

To use a loop invariant to prove correctness, ~~of each iteration~~ we must show three about it:

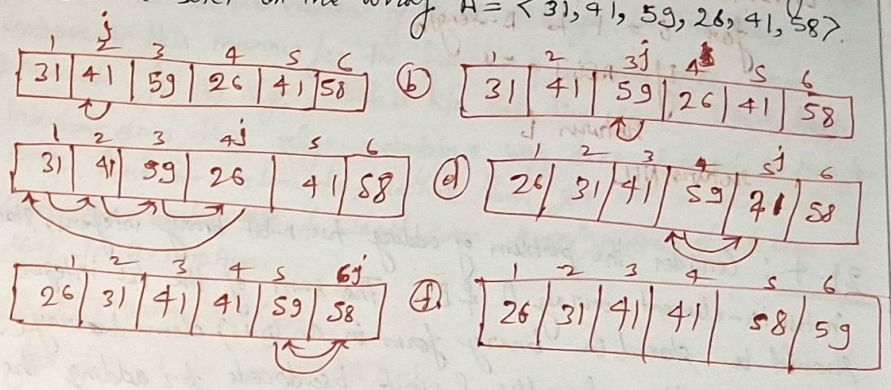
① Initialization: It is true prior to the first iteration of the loop.

② Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

③ Termination: When the loop terminates, the invariant — usually along with the reason that the loop terminated — gives us a useful property that helps show that the algo. is correct.

Exercises:

2.1-1: Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.



2.1-2 Rewrite the INSERTION-SORT procedure to sort into non increasing instead of non-decreasing order.

```
INSERTION-SORT(A)
for j = 2 to A[j]
    key = A[j]
    i = j - 1
    While i > 0 && A[i] < key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

2.1-3: Consider the searching problem:

I/p: A sequence of n numbers $A = \langle a_1, a_2 \ldots a_n \rangle$ & a value v.

o/p: An index i such that $v = A[i]$ or the special value NIL if v does not appear in A.

Write pseudocode for linear search, which scans through the sequence looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

```
LINEAR-SEARCH (A, v)
    for i = 1 to A.length
        if A[i] == v
            return i
    return NIL
```

2.1-4: Consider the problem of adding two n-bit binary integers, stored in two n-element array A & B. The sum of the two integers should be stored in binary form in an (n+1)-element array C. State the problem formally & write pseudocode for adding the two integers.

I/P: A & B integer arrays, A.length == B.length == n
$\forall i \in \{1 \ldots n\}$: A[i], B[i] $\in \{0, 1\}$

O/P: C integer array, C.length == n+1
$\forall i \in \{1 \ldots n+1\}$: C[i] $\in \{0, 1\}$

$$\sum_{i=1}^{n+1} C[i] * 2^{i-1} = \sum_{j=1}^{n} A[j] * 2^{i-1} + \sum_{k=1}^{n} B[k] * 2^{k-1}$$

```
ADD-BINARY (A, B)
    carry = 0
    for i = 1 to A.length
        sum = A[i] + B[i] + carry
        C[i] = sum % 2      // remainder
        carry = sum / 2     // quotient
    C[A.length + 1] = carry ; return C.
```

2.2 : Analysis of Algorithms

2.2-1 : Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ notation. → $\Theta(n^3)$

2.2-2 Consider sorting n numbers stored in array A by first finding the smallest element of A & exchanging it with the element in A[1]. Then find the 2nd smallest element of A & exchange it with A[2]. Continue in this manner for the first n-1 elements of A. Write pseudocode for this algo., which is k/a Selection sort. What loop invariant does this algo. maintain? Why does it need to run for only first n-1 elements, rather than for all n elements? Give the best-case & worst case running times of selection sort in Θ-notation.

```
SELECTION-SORT (A)
1  for i = 1 to Length[A] - 1
2      do min = i
3         for j = i+1 to length[A]
4             do if A[j] < A[min]
5                then min = j
6
7        temp = A[i]
8        A[i] = A[min]      } swapping process.
         A[min] = temp.
```

→ Loop invariant: At start of the loop in line 1, the subarray A[1 . . i-1] consist of smallest i-1 elements in array A with sorted order.

→ After n-1 iterations, the subarray A[1 . . n-1] consists of smallest i-1 elements in array A with sorted order. Therefore, A[n] is already the largest element.

→ Running time : $\Theta(n^2)$.

2.2-3: Consider linear search again (Exercise 2.1-2). How many elements of the I/p sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the avg-case & worst-case running times of linear search in Θ-notation? Justify your ans.

If the element is present in the sequence, half of the elements are likely to be checked before it is found in the avg. case. In the worst case, all of them will be checked. That is $n/2$ checks on the avg. case & n for the worst case. Both of them are $\Theta(n)$.

2.2-4: How can we modify almost any algo. to have a good best-case running time?

You can modify any algo. to have a best-case time complexity by adding a special case. If I/p matches this special case, return the pre-computed answer.

2.3: Designing algorithms

Divide & Conquer:

→ Divide : The problem divided into a no. of subproblems.
→ Conquer : The subproblems by solving them recursively.
→ Combine : The subproblem solutions to give a solution to the original problem.

Merge Sort

→ based on divide & conquer.
→ Its worst-case running time has a lower order of growth than insertion sort. Because, we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p..r]$.
Initially, $p=1$ & $r=n$, but these values change as we recurse through subproblems.

To sort $A[p..r]$:

→ Divide : by splitting into subarrays $A[p..q]$ & $A[q+1..r]$, where q is the halfway point of $A[p..r]$.
→ Conquer : by recursively sorting the two subarrays $A[p..q]$ & $A[q+1..r]$
→ Combine : by merging the two sorted subarrays $A[p..q]$ & $A[q+1..r]$ to produce a single sorted subarray $A[p..r]$. To accomplish this step, we'll define a procedure MERGE (A, p, q, r).

```
MERGE-SORT(A, p, r)
if p < r                                   // check for base case
  then q = ⌊(p+r)/2⌋                       // check for base case  Divide → Θ(1)
       MERGE-SORT(A, p, q)                 // Divide  Conquer   } 2T(n/2)
       MERGE-SORT(A, q+1, r)               // Conquer
       MERGE(A, p, q, r)                   // Combine  → Θ(n)
```

$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T(n/2) + \Theta(n) & \text{if } n>1. \end{cases}$$

Merging : MERGE procedure

Input: Array A & indices p, q, r such that

→ $p \le q < r$

→ subarray $A[p..q]$ is sorted & subarray $A[q+1..r]$ is sorted. By the restrictions on p, q, r, neither subarray is empty.

Output: Two subarrays are merged into a single sorted subarray in $A[p..r]$.

We implement it so that it takes $\Theta(n)$ time, where, $n = r - p + 1$ = the no. of elements being merged.

```
MERGE (A, p, q, r)
n1 = q - p + 1
n2 = r - q
Create arrays L[1..n1+1] & R[1..n2+1]
for i = 1 to n1
    do L[i] = A[p + i - 1]
for j = 1 to n2
    do R[j] = A[q + j]
L[n1+1] = ∞
R[n2+1] = ∞
i = 1
j = 1
for k = p to r                    → Θ(n)
    do if L[i] ≤ R[i]
          then A[k] = L[i]
               i = i + 1
          else A[k] = R[j]
               j = j + 1
```

Solving the merge-sort recurrence

* By the master theorem: $T(n) = a\,T(n/b) + \Theta(n^k \log^p n)$

$a = 2,\ b = 2,\ k = 1 \& p = 0$

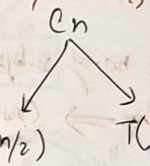
$a = b^k \Rightarrow (2 = 2)$. Average case,

$$T(n) = \Theta(\log_b^{a} \log_n^{p+1}) = \Theta(n \log n).$$

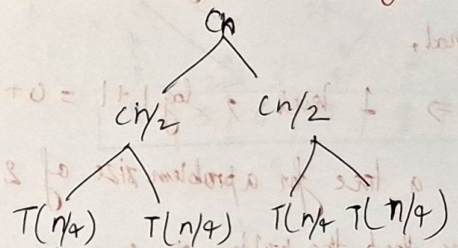
* By the recursion tree method: We rewrite the recurrence as

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n/2) + cn & \text{if } n > 1. \end{cases}$$

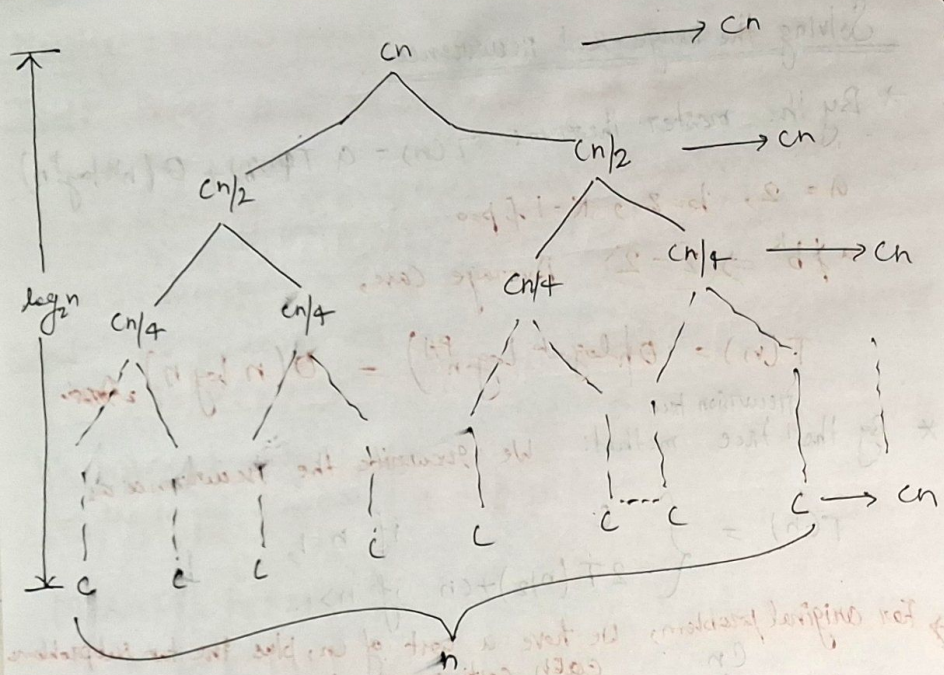
→ For original problem, we have a cost of cn, plus the two subproblems each costing $T(n/2)$:



→ For each of size-$n/2$ subproblems, we have a cost of $cn/2$ plus 2 subproblems, each costing $T(n/4)$.



→ Continue expanding until the problem sizes get down to 1:



→ Each time we go down one level, the no. of subproblems doubles but the cost per subproblem halves ⟹ Cost per level stays the same.

→ There are $(\log n + 1)$ level (height = $\log n$)

→ Use induction method,

- Base case: $n = 1 \Rightarrow$ 1 level, $\log 1 + 1 = 0 + 1 = 1$

- Inductive hypothesis: a tree for a problem size of 2 has $\log 2^i + 1 = i + 1$ levels.

→ Because, we assume that the problem size is a power of 2, next problem size up after 2^i is 2^{i+1}.

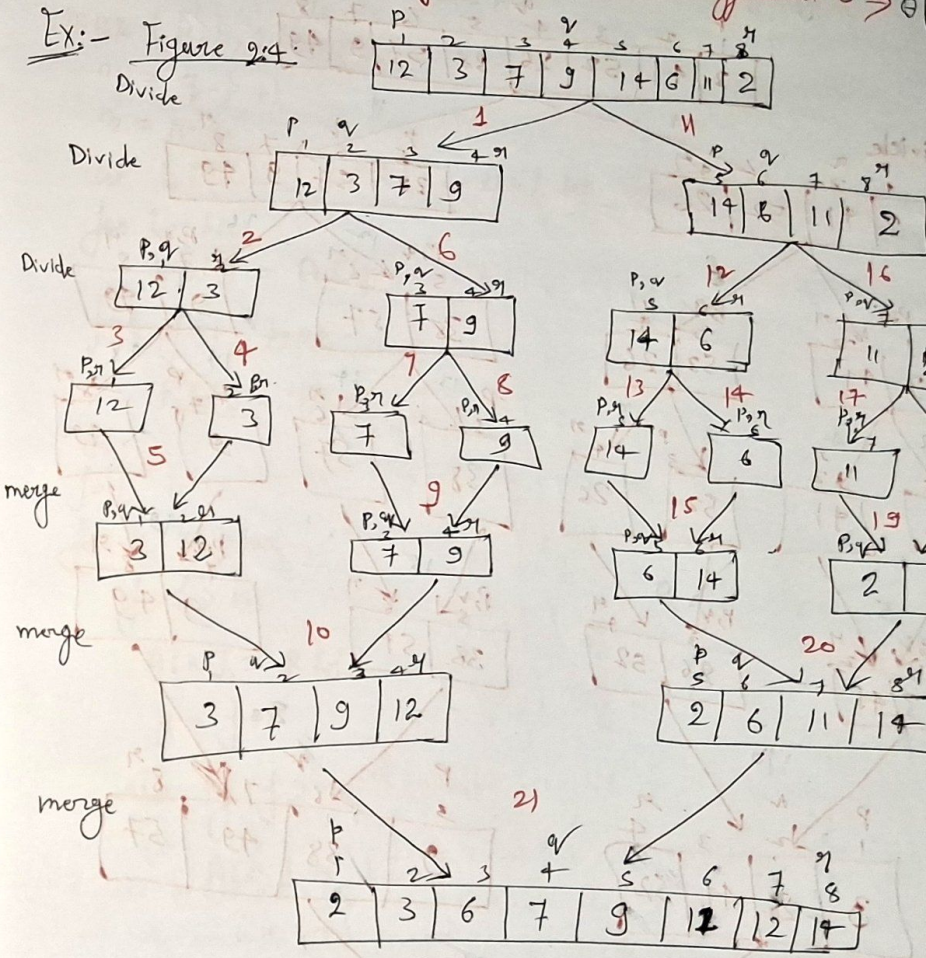
→ A tree for a problem size of 2^{i+1} has one more level than the size-2 tree ⟹ $i+2$ levels.

→ Since $\log 2^{i+1} + 1 = i + 2$, we're done with the inductive argument.

→ Total cost is sum of costs at each level. Have $\log n + 1$ level each costing cn ⟹ Total cost is $cn \log n + cn$.

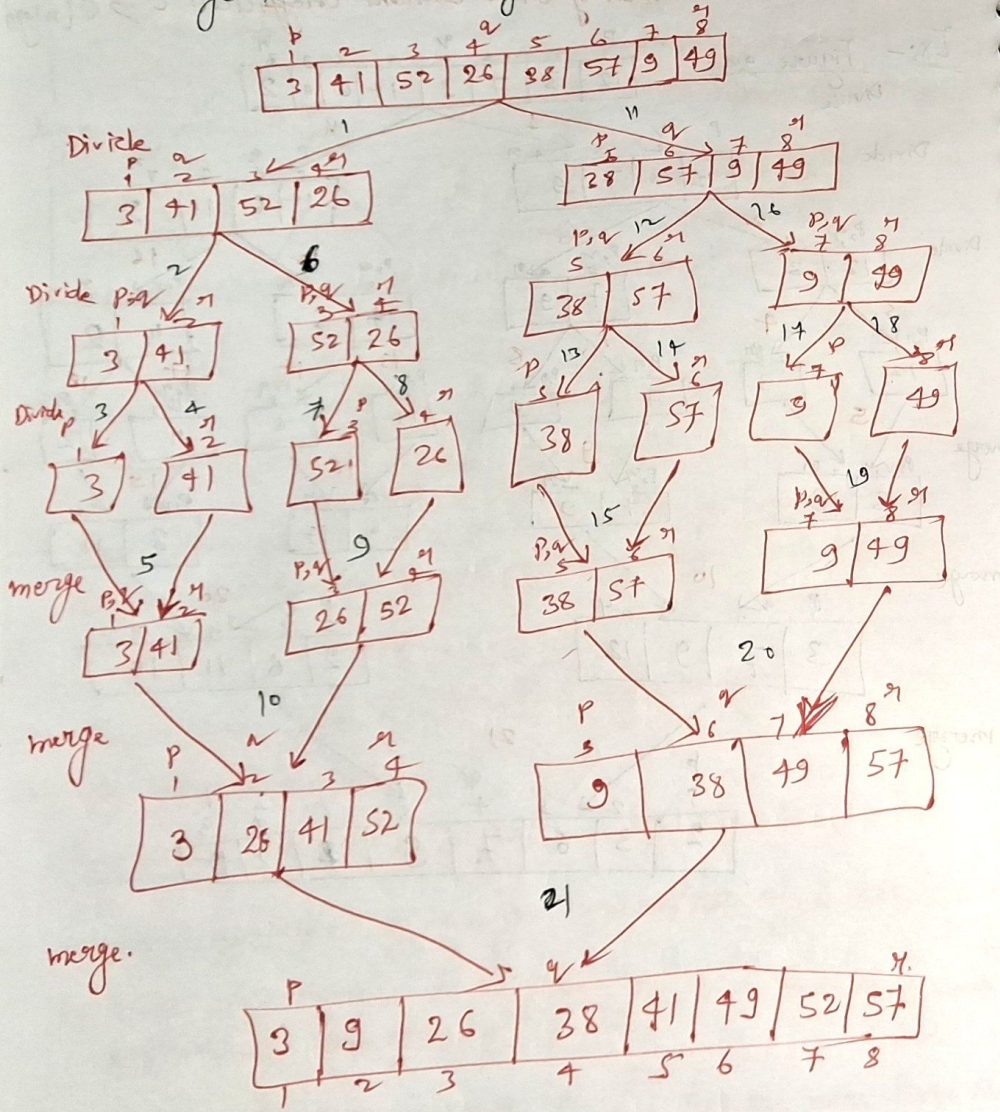
→ Ignore low order term of cn & constant coefficient $c \Rightarrow \Theta(n \log n)$

Ex:- Figure 2.4:



Exercises

2.3-1 Using Figure 2.4 as a model, illustrate the operation of merge sort on the array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.



2.3-2: Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A & then copying the remainder of the other array back into A.

```
MERGE (A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
  let L[1..n1] and R[1..n2] be new arrays.
  for i = 1 to n1
      L[i] = A[p + i - 1]
  for j = 1 to n2
      R[j] = A[q + j]
  i = 1
  j = 1
  for k = p to r
      if i > n1
          A[k] = R[j]
          j = j + 1
      else if j > n2
          A[k] = L[i]
          i = i + 1
      else if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else
          A[k] = R[j]
          j = j + 1.
```

2.3-3 Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2 \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \log n$

Sol^n → Base Case: For $n = 2^1$, $T(n) = 2 \log 2 = 2$.

→ Suppose, $n = 2^k$, $T(n) = n \log n \Rightarrow 2^k \log 2^k = 2^k \cdot k$.

→ For $n = 2^{k+1}$,

$$\begin{aligned} T(n) &= 2T(2^{k+1}/2) + 2^{k+1} \\ &= 2T(2^k) + 2^{k+1} \\ &= 2 \cdot 2^k \cdot k + 2^{k+1} \quad [\because n = 2^k] \\ &= 2^{k+1}(k+1) \\ &= 2^{k+1} \log 2^{k+1} \\ &= n \log n. \end{aligned}$$

Hence proved.

2.3-4: We can express insertion sort as a recursive procedure as follows. In order to sort $A[1..n]$, we recursively sort $A[1..n-1]$ & then insert $A[n]$ into the sorted array $A[1..n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

It takes $\Theta(n)$ times in the worst case to insert $A[n]$ into the sorted array $A[1..n-1]$. Therefore, the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n-1) + \Theta(n) & \text{if } n > 1 \end{cases}$$

The solution of the recurrence is $\Theta(n^2)$.

2.3-5: Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against v & eliminate half of the sequence from further consideration. The binary search algorithm repeats this procedure, having the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\log n)$.

Iterative:

```
ITERATIVE-BINARY-SEARCH (A, v, low, high)        // v → key value / target value
    while low ≤ high
        mid = floor((low + high)/2)
        if v == A[mid]
            return mid
        else if v > A[mid]
            low = mid + 1
        else high = mid - 1
    return NIL
```

Each time we do the comparison of v with the middle element, the search range continues with range halved.

The recurrence,

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n/2) + \Theta(1) & \text{if } n > 1 \end{cases}$$

The sol^n of recurrence is $T(n) = \Theta(\log n)$.

Recursive:

```
RECURSIVE-BINARY-SEARCH (A, v, low, high)
    if low > high
        return NIL
    mid = floor((low + high)/2)
    if v == A[mid]
        return mid
    else if v > A[mid]
        return RECURSIVE-BINARY-SEARCH (A, v, mid+1, high)
    else return RECURSIVE-BINARY-SEARCH (A, v, low, mid-1)
```

2.3-6: Observe that the while loop of lines 5-7 of the INSERTION-SORT procedure in section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1..j-1]$. Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \log n)$?

Each time the while loop of line 5-7 of INSERTION-SORT scans backward through the sorted array $A[1..j-1]$. The loop not only searches for the proper place for $A[j]$, but it also moves each of the array elements that are bigger than $A[j]$ one position to the right (line 6). These movements takes $\Theta(j)$ times which occurs all the $j-1$ elements preceding $A[j]$ are larger than $A[j]$. The running time of using binary search search to search is $\Theta(\log j)$, which is still dominated by the running time of moving element $\Theta(j)$.

Therefore, we can't improve the overall worst-case running time of insertion sort to $\Theta(n \log n)$.

*2.3-7: Describe a $\Theta(n \log n)$-time algo. that, given a set S of n integers & another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

First, Sort S, which takes $\Theta(n \log n)$. Then for each element x_i in S, $i = 1, 2 \ldots n$. search $A[i+1..n]$ for $x' = x - x_i$ by binary search, which takes $\Theta(\log n)$.

→ If x' is found, return its position;

→ otherwise, continue for next iteration.

The time complexity of the algorithm is $\Theta(n \log n) + n \cdot \Theta(\log n)$

i.e. $\Theta(n \log n)$.

Chapter-2 problems

2-1: Insertion Sort on small arrays in merge sort.

Although merge sort runs in $\Theta(n \log n)$ worst-case time & insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to coarsen the leaves of the recursion by using insertion sort within merge sort when subproblems becomes sufficiently small. Consider a modification ~~of than merge~~ to merge sort in which n/k sublists of length k are sorted using insertion & then merged using the standard merging mechanism, where, k is a value to be determined.

(a) Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time.

(b) Show how to merge the sublists in $\Theta(n \log(n/k))$ worst-case time.

(c) Given that the modified algorithm runs in $\Theta(nk + n \log(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge, sort, in terms of Θ-notation?

(d) How should we choose k in practice?

(a) The worst-case time of sort a list of length k by insertion sort is $\Theta(k^2)$. Therefore, sorting n/k sublists, each of length k takes $\Theta(k^2 \cdot n/k) = \Theta(nk)$ worst-case time.

(b) We have n/k sorted sublists each of length k. To merge these n/k sorted sublists to a single sorted list of length n, we have to take 2 sublists at a time & continue to merge them. The process can be visualized as a tree with $\log(n/k)$ levels & we compare n elements in each level. Therefore, the worst-case time to merge the sublists is $\Theta(n \log(n/k))$.

The modified algorithm has time to merge ~~the~~ sort when

$\Theta(nk + n\log(n/k)) = \Theta(n\log n)$.

Assume, $k = \Theta(\log n)$

$$\begin{aligned}\Theta(nk + n\log(n/k)) &= \Theta(nk + n\log n - n\log k)\\ &= \Theta(n\log n + n\log n - n\log(\log n))\\ &= \Theta(2n\log n - n\log(\log k))\\ &= \Theta(n\log n).\end{aligned}$$

(d) choose k be the largest length of sublist on which insertion sort is faster than merge sort.

2-2: Correctness of bubble sort

Bubble sort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are ~~out~~ of order.

```
BUBBLESORT(A)
(1) for i = 1 to A.length - 1
(2)     for j = A.length down to i+1
(3)         if A[j] < A[j-1]
(4)             exchange A[j] with A[j-1].
```

(a) Let A' denote the o/p of BUBBLESORT(A) To prove the it is correct, we need to prove that it terminates & that

$A'[1] \le A'[2] \le \cdots \le A'[n]$.

where $n = A.length$. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

The next part will prove inequality (2.3).

(b) State precisely a loop invariant for the for loop in lines 2-4, prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

(c) Using the termination condition of the loop invariant proved part (b), state a loop invariant for the for loop in lines 1-4 that will allow you to prove inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.

(d) What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

Answer:

(a) A' consists of the ~~elem~~ elements in A but in sorted order.

(b) Loop invariant: At the start of each iteration of the for loop of lines 2-4, the subarray $A[j..n]$ consists of the elements originally in $A[j..n]$ before entering the loop but possibly in a different order & the first element $A[j]$ is the smallest among them.

Initialization: Initially the subarray contains only the last element $A[n]$, which is trivially the smallest element of the subarray.

~~increases by one~~

Maintenance: In every step we compare $A[j]$ with $A[j-1]$ & make $A[j-1]$ the smallest among them. After the iteration, the length of the subarray increases by one & the first element is the smallest of the subarray.

Termination: The loop terminates when $j=i$. According to the statement of loop termination in loop invariant, $A[i]$ is the smallest among $A[i..n]$ & $A[i..n]$ consists of the elements originally in $A[i..n]$ before entering the loop.

(c) **Loop invariant:** At the start of each iteration of the for loop of lines 1-4, the subarray $A[1..i-1]$ consists of the $i-1$ smallest elements in $A[1..n]$ in sorted order. $A[i..n]$ consists of the $n-i+1$ remaining elements in $A[1..n]$.

Initialization: Initially the subarray $A[1..i-1]$ is empty & trivially this is smallest element of the subarray.

Maintenance: For part (b), after the execution of the inner loop, $A[j]$ will be the smallest element of subarray $A[i..n]$ & in the beginning of the outer loop, $A[1..i-1]$ consists of elements that are smaller than the elements of $A[i..n]$, in sorted order. So, after the execution of the outer loop, subarray $A[1..i]$ will consists of elements that are smaller than the elements of $A[i+1..n]$ in sorted order.

Termination: The loop terminates when $i = A.length$. At the point the array $A[1..n]$ will consists of all elements in sorted order.

(d) The running time depends on the number of iterations of the for loop of lines 2-4. For a given value of i, this loop makes $n-i$ iterations, & i takes on the values $1, 2, \ldots, n$.

The total no. of iterations,

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

$$= n^2 - n^2/2 - n/2 = \frac{n^2}{2} - n/2$$

Then $\Theta(n^2)$ in worst-case as in the insertion sort in worst-case time.

3: Growth of functions

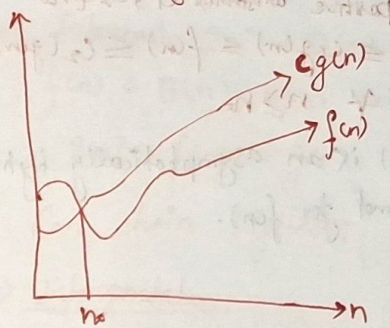
→ A way to describe behaviours of functions in the limit. We are studying asymptotic efficiency.

→ A way to compare "sizes" of functions:

$O \approx \leq$, $\Omega \approx \geq$, $\Theta \approx =$, $o \approx <$, $\omega \approx >$

3.1: Asymptotic notation:

O-notation (Big-oh notation)



$\Rightarrow O(g(n)) = \{ f(n) : \exists$ positive constant c & n_0 such that $0 \leq f(n) \leq cg(n) \; \forall n \geq n_0 \}$

→ $g(n)$ is an asymptotic upper bound for $f(n)$.

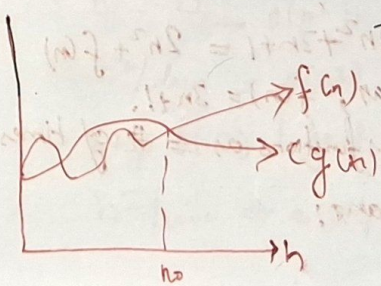
→ If $f(n) \in O(g(n))$ then we write $f(n) = O(g(n))$

Ex:- $\underbrace{2n^2}_{f(n)} = \underbrace{O(n^3)}_{g(n)}$, with $c = 1$ & $n_0 = 2$.

ie $f(n)$ is $O(g(n))$ if there exist $c > 0$ & $n_0 > 0 \Rightarrow f(n) \leq c \cdot g(n)$ $\forall n \geq n_0$.

If $f(n) = n^2 \;/\; n^2+n \;/\; n^2+1000n \;/\; 1000n^2+1000n \;/\; n/1000 \;/\; n^{1.9999} \;/\; n^2/\log\log\log n$) still $f(n) \leq c \cdot g(n)$

Ω-notation



$\rightarrow \Omega(g(n)) = \{ f(n) :$ ∃ positive constants such that $0 \leq cg(n) \leq f$ $\forall n \geq n_0 \}$.

→ $g(n)$ is an asymptotic lower for $f(n)$.

Ex:- $\frac{\sqrt{n}}{f(n)} = \Omega(\underbrace{\log n}_{g(n)})$, with $c=1$ & $n_0=16$.

i.e. $f(n)$ is $\Omega(g(n))$ if there exist $c>0$, $n_0>0$ & $\forall n \geq n_0$

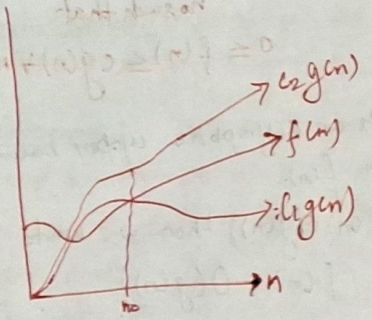
$0 \leq f(n) \geq c\,g(n) \Rightarrow f(n) = \Omega(g(n))$.

If $f(n) = n^2 / n^2+n / n^2-n / 1000n^2+1000n / 1000n^2-1000n / n^3 / n^{2.0001} / n^2 \log\log\log n / 2^{2n} \Rightarrow f(n) = \Omega(g(n))$.

Θ-notation: $\Theta(g(n)) = \{ f(n) : \exists$ positive constants c_1, c_2 & n_0

$\Rightarrow 0 \leq c_1 g(n) \leq f(n) \leq c_2 (g(n))$

$\forall\ n \geq n_0$.



→ $g(n)$ is an asymptotically tight bound for $f(n)$.

Ex:- $n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$ & $n_0 = 8$.

Theorem: $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$.

Leading constants & low-order terms don't matter.

Asymptotic notation in equations

→ When on right-hand side: $\Rightarrow O(n^2)$ stands for some anonymous funct^n in the set $O(n^2)$.

$\Rightarrow 2n^2+3n+1 = 2n^2 + \Theta(n)$ means $2n^2+3n+1 = 2n^2 + f(n)$ for some $f(n) \in \Theta(n)$. In particular, $f(n) = 3n+1$.

$\Rightarrow$ By the way, we interpret # of anonymous functions as = # of times the asymptotic notation appears:

$\sum_{i=1}^{n} O(i)$ Ok: 1 anonymous function

$O(1) + O(2) \cdots + O(n)$ Not Ok: n hidden constants $\Rightarrow$ no clean interpretation.

→ When on left-hand side: $\Rightarrow$ No matter how the anonymous functions are chosen on the left-hand side, there is a way to choose the anonymous functions on the right-hand side to make the equation valid.

$\Rightarrow$ Interpret $2n^2 + \Theta(n) = \Theta(n^2)$ as meaning for all functions $f(n) \in \Theta(n)$, there exists a funct^n $g(n) \in \Theta(n^2) \Rightarrow$ $2n^2 + f(n) = g(n)$.

Can chain together: $2n^2+3n+1 = 2n^2+\Theta(n) = \Theta(n^2)$

$\Rightarrow$ Interpretation:

* First equation: $\exists\ f(n) \in \Theta(n) \Rightarrow 2n^2+3n+1 = 2n^2+f(n)$
* 2nd equation: $\forall\ g(n) \in \Theta(n)$ [$\Rightarrow f(n)$ used to make the 1st equation hold], $\exists\ h(n) \in \Theta(n^2) \Rightarrow 2n^2+g(n) = h(n)$.

o-notation (small-oh): * $o(g(n)) = \{ f(n) :$ $\forall$ constants $c>0$, $\exists\ n_0>0 \Rightarrow 0 \leq f(n) < c \cdot g(n)\ \forall\ n$

* Another view, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

$n^1 \cdot 9999 = o(n^2)$

$n^2/\log n = o(n^2)$

$n^2 \neq o(n^2)$ (just like $2 \not< 2$)

$n^2/1000 \neq o(n^2)$.

ω-notation:

* $\omega(g(n)) = \{ f(n) : \forall\, c>0, \exists\, n_0>0 \Rightarrow 0 \le c\,g(n) < f(n) \;\; \forall\, n \ge n_0 \}$.

* Another view, again probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

$n^{2.001} = \omega(n^2)$

$n^2 \log n = \omega(n^2)$

$n^2 \neq \omega(n^2)$.

Comparisions of functions.

* Relational properties:

① Transitivity: $f(n) = \Theta(g(n)) \;\&\; g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$. (same for $O, \Omega, o \,\&\, \omega$)

② Reflexivity: $f(n) = \Theta(f(n))$ } same for $O \,\&\, \Omega$

③ symmetry: $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$.

④ Transpose symmetry: $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$.
$f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$.

* Comparisons:

→ $f(n)$ is asymptotically smaller than $g(n)$ if $f(n) = o(g(n))$.

→ $f(n)$ is asymptotically larger than $g(n)$ if $f(n) = \omega(g(n))$.

* Trichotomy: For any 2 real numbers $a \,\&\, b$, exactly one of the following must hold: $a<b$, $a=b$ OR $a>b$.

i.e. No trichotomy, Although intuitively, we can liken O to $\le$, Ω to $\ge$, etc., unlike real numbers, where $a<b$, $a=b$ or $a>b$ we must not be able to compare functions.

Ex:- $n^{1+\sin n}$ & n, since $1+\sin n$ oscillates b/w 0 to 2.

Excercises:

3.1-1: Let $f(n) + g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ-notion, prove that $\max(f(n), g(n)) = \Theta(f(n)+g(n))$.

For asymptotically nonnegative functions $f(n) \,\&\, g(n)$, we know that,

$\exists\, n_1, n_2 : f(n) \ge 0 \;\; \forall\, n > n_1$

$g(n) \ge 0 \;\; \forall\, n > n_2$.

Let $n_0 = \max(n_1, n_2)$ & we know that the equations below would be true for $n > n_0$:

$f(n) \le \max(f(n), g(n))$

$g(n) \le \max(f(n), g(n))$

$(f(n) + g(n))/2 \le \max(f(n), g(n))$

$\max(f(n), g(n)) \le (f(n) + g(n))$

Then we can combine last 2 inequalities:

$$0 \le \frac{f(n)+g(n)}{2} \le \max(f(n), g(n)) \le g(n) + f(n).$$

Which is the definition of $\Theta(f(n)+g(n))$ with $c_1 = \frac{1}{2}$ & $c_2 = 1$.

3.1-2: Show that for any real constants a & b, where $b>0$

$$(n+a)^b = \Theta(n^b).$$

Expand $(n+a)^b$ by Binomial expansion, we have

$$(n+a)^b = C_0^b n^b a^0 + C_1^b n^{b-1} a^1 + \cdots + C_b^b n^0 a^b.$$

Besides, we know below is true for any polynomial when $x \geq 1$.

$$a_0 x^0 + a_1 x^1 + \cdots + a_n x^n \leq (a_0 + a_1 + \cdots + a_n) x^n.$$

Thus,

$$C_0^b n^b \leq C_0^b n^b a^0 + C_1^b n^{b-1} a^1 + \cdots + C_b^b n^0 a^b \leq$$
$$\leq (C_0^b + C_1^b + \cdots + C_b^b) n^b = 2^b n^b$$

$$\Rightarrow (n+a)^b = \Theta(n^b).$$

3.1-3: Explain why the stmt, "The running time of algorithm A is at least $O(n^2)$," is meaningless.

$T(n)$: running time of algorithm A. We just care about the upper bound of $T(n)$.

The statement: $T(n)$ is at least $O(n^2)$.

→ Upper bound: Because "$T(n)$ is at least $O(n^2)$", there's no information about the upper bound of $T(n)$.

→ Lower bound: Assume $f(n) = O(n^2)$, then stmt: $T(n) \geq f(n)$ but $f(n)$ could be any fun^n that is "smaller" than n^2. Ex:- constant, n, etc. So there's no conclusion about the lower bound of $T(n)$, too.

∴ The stmt, "The running time of algo. A is at least $O(n^2)$," is meaningless.

3.1-4: Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

$2^{n+1} = 2 \times 2^n$.

We can choose $c \geq 2$ & $n_0 = 0$

$\Rightarrow 0 \leq 2^{n+1} \leq c \times 2^n$

$\forall n \geq n_0$.

By definition $2^{n+1} = O(2^n)$ True

$2^{2n} = 2^n \times 2^n = 4^n$.

We can't find any c & n_0

$\Rightarrow 0 \leq 2^{2n} = 4^n \leq c \times 2^n$

$\forall n \geq n_0$.

Hence, $2^{2n} \neq \Theta(2^n)$ False

3.1-5: Prove the Theorem 3.1.

The theorem states:

For any 2 functions $f(n)$ & $g(n)$, we have $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$.

From $f(n) = \Theta(g(n))$, we have that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n > n_0.$$

We can pick the constants from here & use them in the definition of O & Ω to show that both hold.

From $f(n) = \Omega(g(n))$ & $f(n) = O(g(n))$, we have that

$$0 \leq c_3 g(n) \leq f(n) \quad \forall n \geq n_1$$

& $0 \leq f(n) \leq c_4 g(n) \quad \forall n \geq n_2$

If we let $n_3 = \max(n_1, n_2)$ & merge the inequalities, we get

$$0 \leq c_3 g(n) \leq f(n) \leq c_4 g(n) \quad \forall n > n_3$$

Which is the definition of Θ.

3.1-6: Prove that the running time of an algorithm is $\Theta(g(n))$ iff its worst-case running time is $O(g(n))$ & its best-case running time $\Omega(g(n))$.

If T_w is the worst-case running time & T_b is the best-case running time, we know that

$$0 \le c_1 g(n) \le T_b(n) \quad \forall\ n > n_b$$

& $$0 \le T_w(n) \le c_2 g(n) \quad \forall\ n > n_w.$$

Combining them we get

$$0 \le c_1 g(n) \le T_b(n) \le T_w(n) \le c_2 g(n)$$

$$\forall\ n > \max(n_b, n_w).$$

Since the running time is bound b/w T_b & T_w & the above is the definition of the Θ-notation, proved.

3.1-7: Prove $o(g(n)) \cap \omega(g(n))$ is the empty set.

Let $f(n) = o(g(n)) \cap \omega(g(n))$. We know that for any $c_1 > 0$, $c_2 > 0$.

$$\exists\ n_1 > 0 : 0 \le f(n) < c_1 g(n)$$

& $$\exists\ n_2 > 0 : 0 \le c_2 g(n) < f(n).$$

If we pick $n_0 = \max(n_1, n_2)$ & let $c_1 = c_2$, for the problem definition, we get

$$c_1 g(n) < f(n) < c_1 g(n)$$

There is no sol^n, which means that the intersection is the empty set.

3.1-8: We can extend our notion to the case of two parameters n & m that can go to infinity independently to different rates. For a given funct^n $g(n,m)$ we denote $O(g(n,m))$ the set of functions:

$$O(g(n,m)) = \{ f(n,m) : \exists +ve\ c, n_0 \& m_0$$
$$\Rightarrow 0 \le f(n,m) \le c g(n,m)$$
$$\forall\ n \ge n_0 \& m \ge m_0 \}.$$

Give corresponding definition for $\Omega(g(n,m))$ & $\Theta(g(n,m))$.

$$\Omega(g(n,m)) = \{ f(n,m) : \exists +ve\ c, n_0 \& m_0$$
$$\Rightarrow 0 \le c g(n,m) \le f(n,m)$$
$$\forall\ n \ge n_0 \& m \ge m_0 \}.$$

$$\Theta(g(n,m)) = \{ f(n,m) : \exists +ve\ c_1, c_2, n_0, \& m_0$$
$$\Rightarrow 0 \le c_1 g(n,m) \le f(n,m) \le c_2 g(n,m))$$
$$\forall\ n \ge n_0 \& m \ge m_0.$$

3.2: Standard notations & common functions

* Monotonicity:

→ $f(n)$ is monotonically increasing if $m \leq n \Rightarrow f(m) \leq f(n)$

→ " " " decreasing if $m \geq n \Rightarrow f(m) \geq f(n)$

→ " " strictly increasing if $m < n \Rightarrow f(m) < f(n)$

→ " " " decreasing if $m > n \Rightarrow f(m) > f(n)$.

* Exponentials

Useful identities:

$$a^{-1} = 1/a \quad , \quad (a^m)^n = a^{mn} \quad , \quad a^m a^n = a^{m+n}$$

Can relate rates of growth of polynomials & exponentials:

$\forall$ real constants a & $b \Rightarrow a > 1$,

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0, \Rightarrow \boxed{n^b = o(a^n)}.$$

A suprisingly useful inequality: $\forall$ real x,

$$e^x \geq 1 + x.$$

As x gets closer to 0, e^x gets closer to $1+x$.

* Logarithms

Notations:

$\log_2 n = \lg n$ (binary logarithm),

$\ln n = \log_e n$ (natural logarithm),

$\lg^k n = (\lg n)^k$ (exponentiation),

$\log \lg n = \log(\lg n)$ (composition).

Logarithm functions apply only to the next term in the formula, so that $\lg n + k = (\lg n) + k \neq \lg(n+k)$.

In the expression $\log_b a$:

→ If we hold b constant, then the expression (exp) is strictly increasing as a increases.

→ If we hold a constant, then the exp. is strictly decreasing as b increasing.

Useful identities for all real $a > 0$, $b > 0$, $c > 0$ & n & where logarithm bases are not 1:

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

$$\log_c(ab) = \log_c a + \log_c b,$$

$$\log_b a^n = n \log_b a,$$

$$\log_b a = \frac{\log_c a}{\log_c b},$$

$$\log_b(1/a) = -\log_b a,$$

$$\log_b a = \frac{1}{\log_a b},$$

$$a^{\log_b c} = c^{\log_b a}.$$

Changing the base of a logarithm from one constant to another only changes the value by a constant factor, so we usually don't worry about logarithm bases in asymptotically notation. Convention is to use log within asymptotic notation, unless the base actually matters.

Just as polynomials grow more slowly than exponentials, logarithms grow more slowly than polynomials.

In $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$, substitute $\lg n$ for n & 2^a for a:

$$\lim_{n\to\infty} \frac{\log^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0 \Rightarrow \log^b n = o(n^a)$$

By $a^{\log_b c} = c^{\log_b a}$: $(2^a)^{\lg n} = (n^a)^{\lg 2} = (n^a)$.

* Factorials:

$n! = 1.2.3.n$ Special case : $0! = 1$.

Can use Stirling's approximation,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right).$$

to derive that $\lg(n!) = \Theta(n \log n)$.

Exercises:

3.2-1: Show that if $f(n)$ & $g(n)$ are monotonically increasing functions, then so are the functions $f(n)+g(n)$ & $f(g(n))$ & if $f(n)$ & $g(n)$ are in addition non-negative, then $f(n).g(n)$ is monotonically increasing.

$f(m) \le f(n)$ for $m \le n$

$g(m) \le g(n)$ for $m \le n$,

$\Rightarrow \boxed{f(m) + g(m) \le f(n) + g(n).}$

Which is proved, the first fnct^n.

Then $f(g(m)) \le f(g(n))$ for $m \le n$

This true, since $g(m) \le g(n)$ & $f(n)$ is monotonically ↑.

If both functions are non negative, then we can multiply the two equalities & we get

$$f(m).g(m) \le f(n).g(n).$$

3.2-2 Prove equation (3.16).

$$a^{\log_b c} = a^{\frac{\log_a c}{\log_a b}} = \left(a^{\log_a c}\right)^{\frac{1}{\log_a b}} = c^{\log_b a}$$

3.2-3 Prove equation (3.19). ~~Also~~ Also prove that $n! = \omega(2^n)$ & $n! = o(n^n)$.

$$\lg(n!) = \Theta(n \lg n) \quad \text{——(3.19)}$$

We can use Stirling approximation to prove these three eqns.

For eqn (3.19),

$$\lg(n!) = \lg\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)\right)$$

$$= \lg\left(\sqrt{2\pi n} + \lg\left(\frac{n}{e}\right)^n + \lg\left(1 + \Theta\left(\frac{1}{n}\right)\right)\right.$$

$$= \Theta(\sqrt{n}) + n\lg\left(\frac{n}{e}\right) + \lg\left|\Theta(1) + \Theta\left(\frac{1}{n}\right)\right.$$

$$= \Theta(\sqrt{n}) + \Theta(n \log n) + \Theta\left(\frac{1}{n}\right)$$

$\boxed{\ln(n!) = \Theta(n \lg n)}$ Hence, proved

For $n! = \omega(2^n)$,

$$\lim_{n\to\infty} \frac{2^n}{n!} = \lim_{n\to\infty} \frac{2^n}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\theta\left(\frac{1}{n}\right)\right)}$$

$$= \lim_{n\to\infty} \frac{1}{\sqrt{2\pi n}\left(1+\theta\left(\frac{1}{n}\right)\right)}\left(\frac{2e}{n}\right)^n$$

$$\leq \lim_{n\to\infty}\left(\frac{2e}{n}\right)^n$$

$$\leq \lim_{n\to\infty} \frac{1}{2^n} = 0 \quad [\because n > 4e]$$

For $n! = o(n^n)$,

$$\lim_{n\to\infty} \frac{n^n}{n!} = \lim_{n\to\infty} \frac{n^n}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\theta\left(\frac{1}{n}\right)\right)}$$

$$= \lim_{n\to\infty} \frac{e^n}{\sqrt{2\pi n}\left(1+\theta\left(\frac{1}{n}\right)\right)}$$

$$= \lim_{n\to\infty} \theta\left(\frac{1}{\sqrt{n}}\right)e^n$$

$$\geq \lim_{n\to\infty} \frac{e^n}{c\sqrt{n}} \quad [\text{For some constant } c > 0]$$

$$\geq \lim_{n\to\infty} \frac{e^n}{cn}$$

$$= \lim_{n\to\infty} \frac{e^n}{c} = \infty.$$

* 3.2-4: Is the function $\lceil \lg n \rceil !$ polynomially bounded?
Is the function $\lceil \lg \lg n \rceil !$ polynomially bounded?

Proving that a function $f(n)$ is polynomially bounded is equivalent to proving that $\lg(f(n)) = O(\lg n)$ for following reasons:

→ If $f(n)$ is polynomially bounded, then there exist constants c, k, n_0 such that $\forall\, n \geq n_0$, $f(n) \leq cn^k$.
Hence, $\lg(f(n)) \leq k \cdot c \lg n$, which means that
$\lg(f(n)) = O(\lg n)$.

→ If $\lg(f(n)) = O(\lg n)$, the f is polynomially bounded.

In the following proofs, we will make use of the following two facts:

1. $\lg(n!) = \Theta(n \lg n)$
2. $\lceil \lg n \rceil = \Theta(\lg n)$

→ $\lceil \lg n \rceil !$ is not polynomially bounded because

$$\lg(\lceil \lg n \rceil !) = \Theta(\lceil \lg n \rceil \lg \lceil \lg n \rceil)$$
$$= \Theta(\lceil \lg n \rceil \lg \lg n)$$
$$= \omega(\lg n)$$
$$\neq O(\lg n)$$

→ $\lceil \lg \lg n \rceil !$ is polynomically bounded because

$$\lg(\lceil \lg \lg n \rceil !) = \Theta(\lceil \lg n \lg n \rceil \lg \lceil \lg \lg n \rceil)$$
$$= \Theta(\lg n \lg n \lg \lg \lg n)$$

$$\lg(\lceil \lg\lg n \rceil !) = \Theta((\lg\lg n)^2)$$
$$= o(\lg^2(\lg n))$$
$$= o(\lg n)$$
$$= O(\lg n).$$

The last step above follows from the property that any polylogarithmic function grows more slowly than any positive polynominal function, i.e. that for constants $a, b > 0$, we have $\lg^b n = o(n^a)$.
Substitute $\lg n$ for n and 2 for b and 1 for a, giving

$$\lg^2(\lg n) = o(\lg n).$$

Therefore, $\lg(\lceil \lg\lg n \rceil !) = O(\lg n)$ & so $\lceil \lg\lg n \rceil !$ is polynomially bounded.

*3.2-5: Which is asymptotically larger: $\lg(\lg^* n)$ or $\lg^*(\lg n)$?

We have $\lg^* 2^n = 1 + \lg^* n$

$$\lim_{n\to\infty} \frac{\lg(\lg^* n)}{\lg^*(\lg n)} = \lim_{n\to\infty} \frac{\lg(\lg^* 2^n)}{\lg^*(\lg 2^n)}$$

$$= \lim_{n\to\infty} \frac{\lg(1+\lg^* n)}{\lg^* n}$$

$$= \lim_{n\to\infty} \frac{\lg(1+n)}{n}$$

$$= \lim_{n\to\infty} \frac{1}{1+n}$$

$$= 0$$

Therefore, we have that $\lg^*(\lg n)$ is asymptotically larger.

3.2-6: Show that the golden ratio ϕ & its conjugate $\hat{\phi}$ both satisfy the equation $x^2 = x+1$.

$$\phi^2 = \left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{6+2\sqrt{5}}{4} = 1 + \frac{1+\sqrt{5}}{2} = 1+\phi$$

$$\hat{\phi}^2 = \left(\frac{1-\sqrt{5}}{2}\right)^2 = \frac{6-2\sqrt{5}}{4} = 1 + \frac{1-\sqrt{5}}{2} = 1+\hat{\phi}.$$

3.2-7: Prove by induction that the ith Fibonacci number satisfies the equality $F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$, where ϕ is the golden ratio & $\hat{\phi}$ is its conjugate.

→ Base Case:

For $i=0$: $\frac{\phi^0 - \hat{\phi}^0}{\sqrt{5}} = \frac{1-1}{\sqrt{5}} = 0 = F_0$

For $i=1$: $\frac{\phi^1 - \hat{\phi}^1}{\sqrt{5}} = \frac{(1+\sqrt{5})-(1-\sqrt{5})}{2\sqrt{5}} = 1 = F_1$.

→ Assume,

- $F_{i-1} = (\phi^{i-1} - \hat{\phi}^{i-1})/\sqrt{5}$ &
- $F_{i-2} = (\phi^{i-2} - \hat{\phi}^{i-2})/\sqrt{5}$,

$$F_i = F_{i-1} + F_{i-2}$$
$$= \frac{\phi^{i-1} - \hat{\phi}^{i-1}}{\sqrt{5}} + \frac{\phi^{i-2} - \hat{\phi}^{i-2}}{\sqrt{5}}$$
$$= \left(\phi^{i-2}(\phi+1) - \hat{\phi}^{i-2}(\hat{\phi}+1)\right)/\sqrt{5}$$
$$= \left(\phi^{i-2}\phi^2 - \hat{\phi}^{i-2}\hat{\phi}^2\right)/\sqrt{5} = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$$

3-2-8: Show that $k \ln k = \Theta(n)$ implies $k = \Theta(n/\lg n)$

From symmetry of Θ,

$$k \ln k = \Theta(n)$$
$$\Rightarrow n = \Theta(k \ln k).$$

Let's find $\ln n$,

$$\ln n = \Theta(\ln(k \ln k)) = \Theta(\ln k + \ln \ln k)$$
$$= \Theta(\ln k).$$

Let's divide the two,

$$\frac{n}{\ln n} = \frac{\Theta(k \ln k)}{\Theta(\ln k)} = \Theta\left(\frac{k \ln k}{\ln k}\right) = \Theta(k)$$

3 problems

3-1: Asymptotically behavior of polynomials.

Let $$p(n) = \sum_{i=0}^{d} a_i n^i,$$

where $a_d > 0$, be a degree-d polynomial in n, & let k be a constant. Use the definitions of the asymptotic notations to prove the following properties.

(a) if $k \geq d$, then $p(n) = O(n^k)$.

(b) if $k \leq d$, then $p(n) = \Omega(n^k)$.

(c) if $k = d$, then $p(n) = \Theta(n^k)$.

(d) if $k > d$, then $p(n) = o(n^k)$

(e) if $k < d$, then $p(n) = \omega(n^k)$.

(a) Let's see that $p(n) = O(n^d)$. we need to pick $c = a_d + b$, such that

$$\sum_{i=0}^{d} a_i n^i = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0 \leq c n^d.$$

When we divide by n^d, we get

$$c = a_d + b \geq a_d + \frac{a_{d-1}}{n} + \frac{a_{d-2}}{n^2} + \cdots + \frac{a_0}{n^d}.$$

if $b \geq \frac{a_{d-1}}{n} + \frac{a_{d-2}}{n^2} + \cdots + \frac{a_0}{n^d}$

If we choose $b = 1$, then we can choose n_0,

$$n_0 = \max\left(d a_{d-1}, d\sqrt{a_{d-2}}, \ldots, d\sqrt[d]{a_0}\right).$$

Now, we have no of c, such that

$$p(n) \leq cn^d \text{ for } n \geq n_0.$$

Now we have n_0 & c, such that

$$p(n) \leq cn^d \quad \forall\, n \geq n_0.$$

which is the definition of $O(n^d)$.

By choosing $b = -1$ we can prove the $\Omega(n^d)$ inequality & thus the $\Theta(n^d)$ inequality. It is a very similar to prove the other inequalities.

(b) If $k \leq d$ then $p(n) = \Omega(n^k)$

Since $p(n) = \sum_{i=0}^{d} a_i n^i$ & the leading term is $a_d n^d$, for large n, this leading term will dominate the growth of the polynomials.

→ If $k \leq d$, then the term $a_d n^d$ grows faster than or at least as fast as n^k. This means that for large n, the polynomial will grow at least as fast as n^k, i.e. for some constant c, we can write:

$$p(n) \geq cn^k$$

for sufficiently large n. This holds because $a_d n^d$, the dominant term, grows faster than or equally to n^k when $k \leq d$.

Thus $p(n) = \Omega(n^k)$.

(c) If $k = d$, then $p(n) = \Theta(n^k)$.

When $k=d$, the leading term of the polynomial is $a_d n^d$, & for large n, this term dominates the polynomial. The lower-order terms grow slowly & becomes negligible as n increases.

⇒ we already showed that: $p(n) = O(n^k)$ when $k \geq d$, we also showed that $p(n) = \Omega(n^k)$ when $k \leq d$

Since adding both terms, it follows that

$$p(n) = \Theta(n^k).$$

(d) If $k > d$, then $p(n) = o(n^k)$

Again, the leading term of the polynomial is $a_d n^d$. When $k > d$, the polynomial's highest degree term $a_d n^d$ grows slower than n^k because n^k grows much faster than n^d.

→ For large n, the growth of $p(n)$ will be dominated by $a_d n^d$, & for $k > d$, we have:

$$\frac{p(n)}{n^k} = \frac{a_d n^d + \text{lower-order terms}}{n^k} = \frac{a_d n^d}{n^k} + \text{smaller terms.}$$

$$= \frac{a_d}{n^{k-d}} \to 0 \quad \text{as } n \to \infty$$

Hence, $p(n) = o(n^k)$, meaning the polynomial grows asymptotically slower than n^k.

(c) If $k \leq d$, then $p(n) = \omega(n^k)$.

Since the leading term of polynomial is $a_d n^d$ & $k < d$, the leading term $a_d n^d$ grows much faster than n^k.

→ For large n, we can write:

$$\frac{p(n)}{n^k} = \frac{a_d n^d + \text{lower-order terms}}{n^k} = \frac{a_d n^d}{n^k} + \text{smaller terms}$$

$$= a_d n^{d-k}$$

As $n \to \infty$, $n^{d-k} \to \infty$ because $d > k$.

This shows that $p(n)$ grows asymptotically faster than n^k, so:

$$p(n) = \omega(n^k)$$

Hence proved.

3-2: Relative asymptotic growths

Indicate for each pair of expressions (A, B) in the table below, whether A is O, o, Ω, ω, or Θ of B. Assume that $k \geq 1$, $\epsilon > 0$ & $c > 1$ are constants. Your answer should be in the form of the table with "yes" or "no" written in each box.

A	B	O	o	Ω	ω	Θ
$\lg^k n$	n^ϵ	yes	yes	no	no	no
n^k	c^n	yes	yes	no	no	no
$\sqrt{n}$	$n^{\sin n}$	no	no	no	no	no
2^n	$2^{n/2}$	no	no	yes	yes	no
$n^{\lg c}$	$c^{\lg n}$	yes	no	yes	no	yes
$\lg(n!)$	$\lg(n^n)$	yes	no	yes	no	yes.

3-3: Ordering by asymptotic growth rates

(a) Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, \ldots, g_{30}$ of the functions $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3), \ldots, g_{29} = \Omega(g_{30})$. Partition your list into equivalence classes such that functions $f(n)$ & $g(n)$ are in the same class if $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\lg\lg n$	$\lg^* n$	$n \cdot 2^n$	$n \lg\lg n$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

(b) Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part (a), $f(n)$ is neither $O(g_i(n))$ nor $\Omega(g_i(n))$

Much of the following above ranking is based on the below properties:

→ Exponential functions grow faster than polynomial functions, which grow faster than polylogarithmic functions.

→ The base of logarithm doesn't matter asymptotically, but the base of an exponential & degree of a polynomial do matter.

We have the following identities:

① $(\lg n)^{\lg n} = n^{\lg\lg n}$ because $a^{\log_b c} = c^{\log_b a}$

② $4^{\lg n} = n^2$ because $a^{\log_b c} = c^{\log_b a}$

③ $2^{\lg n} = n$.

④ $2 = n^{1/\lg n}$ by raising identity 3 to the power $1/\lg n$.

⑤ $2^{\sqrt{2\lg n}} = n^{\sqrt{2/\lg n}}$ by raising identity 4 to the power $\sqrt{2\lg n}$.

⑥ $(\sqrt{2})^{\lg n} = \sqrt{n}$ because $(\sqrt{2})^{\lg n} = 2^{(1/2)\lg n} = 2^{\lg\sqrt{n}} = \sqrt{n}$.

⑦ $\lg^*(\lg n) = (\lg^* n) - 1$.

The following justifications explain some of the rankings:

① $e^n = 2^n (e/2)^n = \omega(n2^n)$, since $(e/2)^n = \omega(n)$.

② $(\lg n)! = \omega(n^3)$ by taking logs: $\lg((\lg n)!) = \Theta(\lg\lg\lg n)$ by Stirling's approximation; $\lg(n^3) = 3\lg n \cdot \lg\lg n = \omega(3)$.

③ $(\sqrt{2})^{\lg n} = \omega(2^{\sqrt{2\lg n}})$ by taking logs: $\lg(\sqrt{2})^{\lg n} = (\frac{1}{2})\lg n$,
$\lg 2^{\sqrt{2\lg n}} = \sqrt{2\lg n} \cdot (\frac{1}{2})\lg n = \omega(\sqrt{2\lg n})$

④ $2^{\sqrt{2\lg n}} = \omega(\lg^2 n)$ by taking logs: $\lg 2^{\sqrt{2\lg n}} = \sqrt{2\lg n}$,
$\lg\lg^2 n = 2\lg\lg n \cdot \sqrt{2\lg n} = \omega(2\lg\lg n)$.

⑤ $\ln\ln n = \omega(2^{\lg^* n})$ by taking logs: $\lg 2^{\lg^* n} = \lg^* n$.
$\lg\ln\ln n = \omega(\lg^* n)$

⑥ $\lg(n!) = \Theta(n\lg n)$ (equation 3.18)

⑦ $n! = \Theta(n^{n+1/2} e^{-n})$ by dropping constants & low-order terms in equation (3.17).

⑧ $(\lg n)! = \Theta((\lg n)^{\lg n + 1/2} n^{-\lg e})$ because $a^{\log_b c} = c^{\log_b a}$.

$2^{2^{n+1}} > 2^{2^n} > (n+1)! > n! > e^n > n \cdot 2^n > 2^n >$
$(3/2)^n > (\lg n)^{\lg n} = n^{\lg\lg n} > (\lg n)! > n^3 > n^2$
$> n\lg n \& \lg(n!) > n = 2^{\lg n} > (\sqrt{2})^{\lg n} (= \sqrt{n}) >$
$2^{\sqrt{2\lg n}} > \lg^2 n > \ln n > \sqrt{\lg n} > \ln\ln n > 2^{\lg^* n} >$
$\lg^* n \& \lg^*(\lg n) > \lg(\lg^* n) > n^{1/\lg n} (= 2 \& 1)$.

(b) The following $f(n)$ is non-negative & for all functions $g_i(n)$ in part (a), $f(n)$ is neither $O(g_i(n))$ nor $\Omega(g_i(n))$.

$$f(n) = \begin{cases} 2^{2^{n+2}} & \text{if } n \text{ is even;} \\ 0 & \text{if } n \text{ is odd.} \end{cases}$$

3-4 : Asymptotic notation properties

Let $f(n)$ & $g(n)$ by asymptotically positive functions. Prove or disprove each of the following conjectures.

(a) $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.

Disprove, $n = O(n^2)$, but $n^2 \neq O(n)$.

(b) $f(n) + g(n) = \Theta(\min(f(n), g(n)))$

Disprove, $n^2 + n \neq \Theta(\min(n^2, n)) = \Theta(n)$.

(c) $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n)))$, where $\lg(g(n)) \geq 1$ & $f(n) \geq 1$ for sufficiently large n.

Prove, because $f(n) \geq 1$ after a certain $n \geq n_0$
$\exists c, n_0 : \forall n \geq n_0, 0 \leq f(n) \leq c\, g(n)$.

$\Rightarrow 0 \le \lg f(n) \le \lg(c g(n)) = \lg c + \lg g(n)$.

We need to prove that

$$\lg(f(n)) \le d \lg g(n)$$

We can find d,

$$d = \frac{\lg c + \lg g(n)}{\lg g(n)} = \frac{\lg c}{\lg g(n)} + 1 \le \lg c + 1$$

where, the last step is valid, because $\lg g(n) \ge 1$.

(a) $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$.

Disprove, because $2n = O(n)$, but $2^{2n} = 4^n \neq O(2^n)$.

(c) $f(n) = O((f(n))^2)$

Prove, $0 \le f(n) \le c f^2(n)$ is trivial when $f(n) \ge 1$, but if $f(n) < 1$ $\forall n$, it's not correct. However, we don't care this case.

(f) $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$.

Prove, from the first, we know that $0 \le f(n) \le c g(n)$ & we need to prove that $0 \le d f(n) \le g(n)$, which is straightforward with $d = 1/c$.

(g) $f(n) = \Theta(f(n/2))$

Disprove, let's pick $f(n) = 2^n$, we will need to prove that

$\exists c_1, c_2, n_0 : \forall n \ge n_0, 0 \le c_1 \cdot 2^{n/2} \le 2^n \le c_2 \cdot 2^{n/2}$,

which is obviously untrue.

(h) $f(n) + o(f(n)) = \Theta(f(n))$.

Prove that let $g(n) = o(f(n))$. Then

$\exists c, n_0 : \forall n \ge n_0, 0 \le g(n) < c \cdot f(n)$.

We prove that,

$\exists c_1, c_2, n_0 : \forall n \ge n_0, 0 \le c_1 f(n) \le f(n) + g(n) \le c_2 f(n)$.

Thus, if we pick $c_1 = 1$ & $c_2 = c+1$, it holds.

3-5 : Variations on O & Ω

Some authors define Ω in a slightly different way than we do; let's use Ω^∞ (read "omega infinity") for this alternative definition. We say that $f(n) = \Omega^\infty(g(n))$ if there exists a positive constant c such that $f(n) \ge c g(n) \ge 0$ for infinity many integers n.

(a) Show that for any two functions f(n) & g(n) that are asymptotically nonnegative, either $f(n) = O(g(n))$ or $f(n) = \Omega^\infty(g(n))$ or both, whereas this is not true if we use Ω in place of Ω^∞.

We have

$$f(n) = \begin{cases} O(g(n)) \text{ \& } \Omega^\infty(g(n)) & \text{if } f(n) = \Theta(g(n)), \\ O(g(n)) & \text{if } 0 \le f(n) \le c g(n), \\ \Omega^\infty(g(n)) & \text{if } 0 \le c g(n) \le f(n), \text{ for infinity many integers } n. \end{cases}$$

if there are only finite n such that $f(n) \geq c(g(n)) \geq 0$.
when $n \to \infty$, $0 \leq f(n) \leq cg(n)$ i.e. $f(n) = O(g(n))$.
Obviously, it's not hold when we use Ω in place of Ω^∞.

(b) Describe the potential advantages & disadvantages of using Ω^∞ instead of Ω to characterize the running times of programs.
Some authors also define O in a slightly diff. manner; let's use O' for the alternative definition. We say that $f(n) = O'(g(n))$ iff. $|f(n)| = O(g(n))$.

*Advantage: We can characterize all the relationship b/w all functions.

*Disadvantage: We cann't characterize precisely.

(c) What happens to each direction of the "if & only if" in Theorem 3.1 if we substitute O' for O but we still use Ω?
Some authors define $\tilde{O}$ (read "soft Oh") to mean O with logarithmic factors ignored:

$$\tilde{O}(g(n)) = \{ f(n) : \exists \text{ +ve } c, k \text{ \& } n_0 \Rightarrow 0 \leq f(n) \leq cg(n)\lg^k(n) \ \forall\ n \geq n_0 \}.$$

For any 2 funct^{ns} $f(n)$ & $g(n)$, we have if $f(n) = \Theta(g(n))$ then $f(n) = O'(g(n))$ & $f(n) = \Omega(g(n))$ &
$f(n) = \Omega(g(n))$
But the conversion is not true.

(d) Define $\tilde{\Omega}$ & $\tilde{\Theta}$ in a similar manner. Prove the corresponding analog to Theorem 3.1.
We have

$$\tilde{\Omega}(g(n)) = \{ f(n) : \exists c, k \text{ \& } n_0 \Rightarrow 0 \leq cg(n)\lg^k n \leq f(n) \ \forall\ n \geq n_0 \}.$$

$$\tilde{\Theta}(g(n)) = \{ f(n) : \exists c_1, c_2, k \text{ \& } n_0 \Rightarrow 0 \leq c_1 g(n)\lg^k(n) \leq f(n) \leq c_2 g(n)\lg^k(n) \ \forall\ n \geq n_0 \}.$$

For any 2 functions $f(n)$ & $g(n)$, we have $f(n) = \tilde{\Theta}(g(n))$ iff. $f(n) = \tilde{O}(g(n))$ & $f(n) = \tilde{\Omega}(g(n))$.

3-6: Iterated functions

For each of the following functions $f(n)$ & constants c, give as tight a bound as possible on $f_c^*(n)$.

	$f(n)$	c	f_c^*
(a)	$n-1$	0	$\Theta(n)$
(b)	$\lg n$	1	$\Theta(\lg^* n)$
(c)	$n/2$	1	$\Theta(\lg n)$
(d)	$n/2$	2	$\Theta(\lg n)$
(e)	$\sqrt{n}$	2	$\Theta(\lg\lg n)$
(f)	$\sqrt{n}$	1	does not converge
(g)	$n^{1/3}$	2	$\Theta(\log_3 \lg n)$
(h)	$n/\lg n$	2	$\omega(\lg\lg n), o(\lg n)$