1. $3n^2+5n+\log_2 n = O(n^2)$

By definition 2, it states that a function $f(n)$ belongs to $O(g(n))$ if there exists positive constants c and n_0 such that, $f(n) \leq c\,(g(n))$ for all $n \geq n_0$

We have, $f(n) = 3n^2+5n+\log_2 n$

if, $f(n) = 3n^2+5n+\log_2 n$ and $g(n) = n^2$

So, $3n^2+5n+\log_2 n \leq cn^2 \Rightarrow 3n^2+5n+\log_2 n = 3n^2+5n^2+n^2$

$\Rightarrow 3n^2+5n+\log_2 n \leq 9n^2$ [To establish upper bound]

Now, choosing, $c=9$ and $n_0 = 1$, to write

$3n^2+5n+\log_2 n \leq 9n^2$ for all $n \geq 1$

By definition 2, we write that, $3n^2+5n+\log_2 n = O(n^2)$

Using definition 3, states that $f(n) = \Theta(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is bounded by positive constants c_1 and c_2, $0 < c_1 \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c_2$

$f(n) = 3n^2+5n+\log_2 n$, $g(n) = n^2$

To compute: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3n^2+5n+\log_2 n}{n^2} = \lim_{n\to\infty} \left(3+\frac{5}{n}+\frac{\log_2 n}{n^2}\right)$

$= 3+0+0$ [$\frac{5}{n} \to 0$ as $n\to\infty$, $\frac{\log_2 n}{n} \to 0$ as $n\to\infty$]

Since, 3 is a finite constant, by definition 3

= wrong

Given function, $g(n) = 3n^2+5n+\log_2 n$

By def^n 2, we want to show that $g(n) = O(n^2)$ meaning there exist constants $c>0$, n_0 such that: $g(n) \leq c\,f(n)$ for all $n \geq n_0$

where $f(n) = n^2$.

To upper bound each term,

$3n^2$ is already in $O(n^2)$

$5n \leq 5n^2$ for $n \geq 1$, so its in $O(n^2)$

$\log_2 n \leq n^2$ for sufficiently large n

Thus for large n, we can write,

$g(n) = 3n^2+5n+\log_2 n \leq 3n^2+5n^2+n^2 = 9n^2$

for $n \geq 1$, $g(n) \leq 9n^2$

we can choose, $c=9$, $n_0=1$

Hence, by definition 2,

$g(n) = O(n^2)$

Definition 3 states, $g(n) = O(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty \rightarrow c$ for some $c \in R^+$

To compute, $\lim_{n\to\infty} \frac{g(n)}{n^2} = \lim_{n\to\infty} \frac{3n^2+5n+\log_2 n}{n^2} = \lim_{n\to\infty} \left(\frac{3n^2}{n^2} + \frac{5n}{n^2} + \frac{\log_2 n}{n^2}\right)$

$\lim_{n\to\infty} \frac{g(n)}{n^2} = 3+0+0 = 3$ $\left[\frac{3n^2}{n^2} = 3, \frac{5}{n} \to 0 \text{ as } n\to\infty, \frac{\log_2 n}{n^2} \to 0 \text{ as } n\to\infty\right]$

Since 3 is a finite constant, by def^n 3, we conclude, $g(n) = O(n^2)$

2. Tree sort: works by inserting elements into a Binary Search Tree (BST) and then performing an in-order traversal to retrieve the elements in sorted order. Average case time complexity: $O(n\log n)$, In a balanced BST, each iteration takes $O(\log n)$, leading to a total of $O(n\log n)$ for n elements.

Worst case time complexity $O(n^2)$ and worst case space complexity is $O(n)$. as storing elements in sorted order the BST degenerates into a linked list, resulting in $O(n)$ space usage.

Odd-Even Transposition Sort: a parallel sorting algorithm that repeatedly compares and swaps adjacent elements in two alternating phases. Average case time complexity: $O(n^2)$ → the algorithm is based on a pairwise comparison and requires $O(n)$ passes in the worst case. Worst case co space coplexity is $O(1)$ because sorting happens in place without requiring extra memory.

Quicksort: is a divide and conquer sorting algorithm that selects a pivot partitioning the array into elements smaller and greater than the pivot and recursively sorts the subarrays. Average case time copiplexity is $O(n\log n)$ performs $O(\log n)$ levels of recursion where each level involves $O(n)$ work. Worst case space complexity $O(n)$ with already sorted input with largest or smallest pivot chosen. can be improved to $O(\log n)$ using optimization.

Jutification: The avg case time complexity of tree sort arises from the expected height of a randomly constructed BST. which is $O(\log n)$. The depth of the tree remains logarithmic, ensuring efficient insertion and traversal.

Odd-even transposition is similar to bubble sort that requires $O(n^2)$ in average case. In the case of quicksort, if the pivot is selected efficiently, it would be $O(n \log n)$

3. (i) Linear insertion sort over binary insertion sort: When sorting a nearly sorted or small array in an online manner. Linear insertion sort has a best case time complexity of $O(n)$ when the input is nearly sorted. Binary insertion sort improves searching time using binary search $O(\log n)$ but the actual insertion still takes $O(n)$. If elements arrive one by one, linear insertion sort is preferable.

(ii) Heapsort over Quicksort: When worst case performance consistency and inplace sorting are required. Heapsort guarantees $O(n \log n)$ worst case time complexity unlike quicksort which can be $O(n^2)$ in worst case. Also, heapsort requires $O(1)$ space where quicksort may require $O(n)$ space.

(iii) Quicksort over mergesort: When sorting an array in memory with good cache performance. Quicksort is faster in practice due to better cache locality. Mergesort has $O(n \log n)$ complexity always but requires $O(n)$ extra space, making it less suitable for in memory sorting. Quicksort is preferable for general purpose sorting when space efficiency is important

(iv) Tournament sort over sorting by ranking: When the sorted array needs to be dynamically updated and to find a small subset of top-ranked elements. Tournament maintains a structure where new elements can be inserted and sorted efficiently unlike sorting by ranking which requires complete re-ranking when a new element arrives. mostly used in sports tournaments, gaming leaderboards.

(v) Odd-even over bubble: When parallel processing is possible. Odd-even can be efficiently parallelized. Its suitable for multithreaded architecture. Bubblesort requires $O(n^2)$ even when parallelism is available. as bubble sort is inherently sequencial.

4. Time complexity measures how the running time of an algorithm grows with input size n. It depends on the specific algorithm used to solve a problem. For example, Binary search has $O(\log n)$ time complexity while sequential search has $O(n)$. On the other hand, problem complexity represents the best possible that any algorithm can achieve for a given problem. For example, the problem complexity of searching in a sorted list is $\Omega(\log n)$, meaning no algorithm can do better than $O(\log n)$ in the worst case.

Sequential search checks each element of list one by one. Even in a sorted list, it takes $O(n)$ in worst case. $A(n) = O(n) \approx n/2$, $W(n) = O(n)$

Jump search involves jumping by a fixed interval and then performing sequential search within a small range. In that case, $W(n) = \frac{n}{i} + (i-1)$ but its $O(n)$ because of fixed i. $\frac{d}{di}(W(n)) = -\frac{n}{i^2} + 1 \Rightarrow 0 = -\frac{n}{i^2} - 1 \Rightarrow i = \sqrt{n}$

Hence, $W(n) = O(\sqrt{n})$ is the best possible with the approach $[W(n) = \frac{n}{\sqrt{n}} + (\sqrt{n}-1)]$

Recursive partitioning partitions into k intervals of size n/k each. $w(n) = n/i + (i-1) = \frac{n/n}{k} + \frac{n}{k} - 1 = k + \frac{n}{k} - 1 = O(n/k)$. After applying partition recursively we get, $W(n) = k + W(n/k) = k - 1 + W(n/k)$. In general, for any fixed k; $W(n) = (k-1)\log_k n + 1$. Binary seach - by dividing the list into two equal halves at each step, binary search significantly improves search efficiency. $\frac{dW}{dk} = \frac{d}{dk}(k-1)\log_k n + 1 = (k-1)\log_e n(-1)(\log_e k)^{-2}\frac{1}{k} + \frac{\log_e n}{\log_e k}$

$= -\frac{k-1}{k}\frac{\log_e n}{(\log_e k)^2} + \frac{\log_e n}{\log_e k} \Rightarrow 0 = \frac{dW}{dk} \Rightarrow \frac{k-1}{k} = \log_e k \Rightarrow \log_e k = 1 - \frac{1}{k} < 1$

$\Rightarrow k < e^1 = 2.7 \Rightarrow k = 2$ [cannot be 1 & greater than 2]

$\frac{d^2W}{dk^2}$ at $k = 2$, is > 0 for minimum value. Thus $k = 2$ is best. So dividing in more than 2 partition is not better. $W(n) = k - 1 + W(n/k) = 2 - 1 + W(n/2)$.

$= \log_2 n + 1 \Rightarrow W(n) = \log_2 n + 1$

Decision tree represent the sequence of comparisons needed to search an element in a list. Let, $n = 16$, for sequential, $W(n) = n = 16$; jump $\rightarrow W(n) = 2\sqrt{n} - 1 = 2 \cdot 4 - 1 = 7$. for binary $\rightarrow W(n) = \log_2 n + 1$

$= 4 + 1 = 5$

(d) To ensure a maximum stack depth of $\Theta(\log n)$, while maintaining the $O(n \log n)$ expected running time of the algorithm, we need to make sure to perform the recursive operation on the smaller subarray and iterate on the larger subarray.

The modified TRE-QUICKSORT will be:

```
TRE-QUICKSORT-MOD (A, p, r)
  while p < r:
      q = PARTITION (A, p, r)
      if (q-p < r-q):
          TRE-QUICKSORT-MOD (A, p, q-1)
          p = q+1
      else:
          TRE-QUICKSORT-MOD (A, q+1, r)
          r = q-1
```

This modified algorithm applies recursion on the smaller portion and iterates over the larger one in each iteration. Since, each recursive call operates on half the elements (at most), the depth follows a logarithmic pattern. The runtime will be,

$$O(n \log n) + O(\log n)? = O(n \log n)$$

Therefore, keeping the running time at $O(n \log n)$, the worst case stack depth for the modified algorithm will be $\Theta(\log n)$.

5. Let $D(n)$ be the recursion depth for quicksort on a list of n integers. The recurrence for the maximum stack depth is

$D(n) = D(n/2) + 1 = D(n/4) + 2 = D(n/8) + 3 \ldots\ldots = D(n/2^k) + k$

The recursion bottoms out when: $\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$

$D(n) = D(1) + \log_2 k \Rightarrow D(n) = O(\log_2 n)$

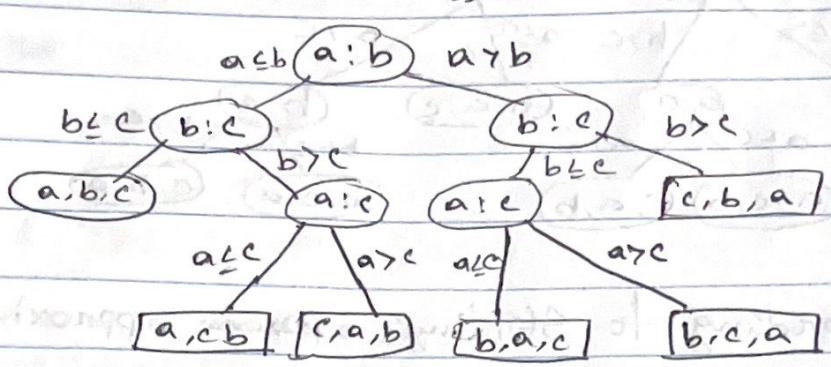
6. let, an input = a, b, c ; There are 3! outputs (for n numbers there are n! outputs). Every decision tree to sort n numbers must have at least n! leaf nodes. Every decision tree to sort n numbers must have $2n! - 1$ nodes. Every decision tree to sort n numbers must have depth at least $\log_2 n!$ leaf nodes. Now, depth is the lower bound worst case time complexity for this class of algorithms.

$\log_2 n! = \log_2 (n * (n-1) * (n-2) \ldots . 1)! = \log_2(n) + \log_2(n-1) + \log_2(n-2) + \ldots + \log_2(1)$

$= \sum_{j=1}^{n} \log_2 j$

$\leq \int_1^n \log_2 x \, dx = (x \log x - x) \Big|_1^n = n \log n - n$

Now, Decision tree for 3 numbers,



∴ Depth of binary tree with n! leaves!

$\geq \log_2 n!$

$\geq \int_1^n \log x\,dx$

$\geq (x\log x - x)\Big|_1^n$

$\geq n\log n - n$

Thus, the weight of decision tree satisfies $\Omega(n\log n)$. A lower bound on problem complexity of comparison based sorting algorithms is $\Omega(n\log n)$

HW-2

Abu Noman Md Sakib
UOM 581

Answer to the Question No-1:

To setup the recurrence relation for the worst-case time complexity of a 3-way search, we need to first analyze the process. The pseudocode for 3-way search is given below:

```
function threeWaySearch(arr, left, right, key):
    if left > right:
        return -1; // key not found.

    mid1 = left + (right - left)/3
    mid2 = right - (right - left)/3

    if arr[mid1] == key:
        return mid1
    if arr[mid2] == key:
        return mid2

    if key < arr[mid1]:
        return threeWaySearch(arr, left, mid1-1, key)
    else if key > arr[mid2]:
        return threeWaySearch(arr, mid2+1, right, key)
    else:
        return threeWaySearch(arr, mid1+1, mid2-1, key)
```

the function splits the input array into three parts and recursively searches in the appropriate segment.

Now, each iteration makes two comparisons at positions $n/3$ and $2n/3$. The function then recursively searches one of the three parts, each of size $n/3$. So, the recurrence relation is:

$$T(n) = T(n/3) + O(1)$$

where, $T(n)$ is the time complexity and $O(1)$ refers to the comparisons.

We solve the recurrence using the iterative expansion method as shown below:

$$T(n) = T(n/3) + O(1)$$
$$T(n/3) = T(n/9) + O(1)$$

Therefore, $T(n) = T(n/9) + O(1) + O(1)$
$$= T(n/9) + 2O(1)$$

$$T(n/9) = T(n/27) + O(1)$$

Similarly, $T(n) = T(n/27) + 3O(1) = T(n/3^3) + 3O(1)$

So, after k iterations, $T(n) = T(n/3^k) + kO(1)$

The recursion stops when $n/3^k = 1$, solving that,

$$n/3^k = 1$$

$$\Rightarrow 3^k = n$$

$$\Rightarrow k = \log_3 n$$

Therefore, $T(n) = O(\log_3 n)$

So, the worst case time complexity of 3-way search is $O(\log_3 n)$

Answer to the Question No-2:

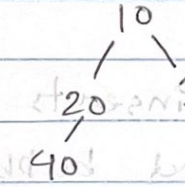
a) To compare if BUILD-MAX-HEAP and BUILD-MAX-HEAP' creates the same heap, we need iterate through both processes. BUILD-MAX-HEAP constructs a max-heap by starting from the last non-leaf node and calling MAX-HEAPIFY to push elements down in the heap. BUILD-MAX-HEAP' constructs the heap incrementally by inserting each element one at a time using MAX-HEAP-INSERT.

BUILD-MAX-HEAP and BUILD-MAX-HEAP' do not always create the same heap for the same input array.

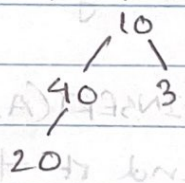
Let, an input array → [10, 20, 30, 40].

Using BUILD-MAX-HEAP,

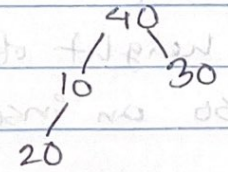
Initial:



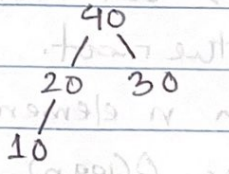
MAX-HEAPIFY (A,2) on 20:



MAX-HEAPIFY (A, 1) on 10:

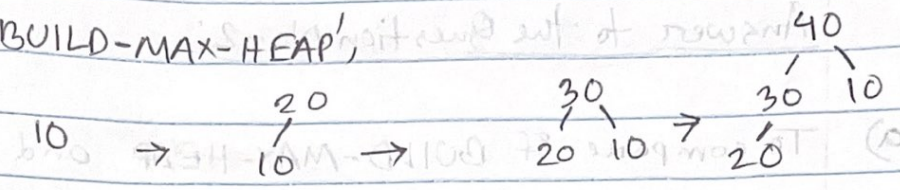


MAX-HEAPIFY (A,2) on 10:



Final heap → [40, 20, 30, 10]

Using BUILD-MAX-HEAP',



final heap → [40, 30, 10, 20]

So, BUILD-MAX-HEAP produced [40, 20, 30, 10] while BUILD-MAX-HEAP' produced [40, 30, 10, 20]. Thus, it is proven that, BUILD-MAX-HEAP and BUILD-MAX-HEAP' do not always generate the same heap.

b) To analyze the time complexity of BUILD-MAX-HEAP', we need to understand that the algorithm calls MAX-HEAP-INSERT n times, where each call inserts an element and maintains the heap properly.

MAX-HEAP-INSERT(A, A[i], n) → inserts an element at the end of the heap and bubbles it up to maintain the heap property. In worst case, an element moves from the last level to the root. The maximum height of a heap with n elements is O(log n), so an insertion takes O(log n) time.

Therefore,

$$T(n) = \sum_{i=1}^{n} O(\log i)$$

$$\sum_{i=1}^{n} \log i = \int_{1}^{n} \log x \, dx$$

$$= [x \log x]_1^n - [x]_1^n$$

$$= n \log n - n + 1$$

$$\approx n \log n \rightarrow O(n \log n)$$

Thus, BUILD-MAX-HEAP' runs in $\theta(n \log n)$ to build an n-element heap in the worst case scenario.

[showed]

Answers to the Question No-3:

a) From the section 7.1, which contains two recursive calls to itself, it is proven that the QUICKSORT algorithm correctly sorts the array A. TRE-QUICKSORT differs from QUICKSORT in only the last line of the loop. In the TRE-QUICKSORT, we eliminated the second recursive call and replaced it with an iterative approach.

if $p \geq r$, the function terminates
the PARTITION(A,p,r) → rearranges the element such that elements in A[p:q-1] are less than or equal to the pivot and A[q+1:r] are greater than or equal to the pivot.
The function recursively sorts the left subarray and then updates the p value.

Each iteration correctly partitions and sorts the left subarray, then updates p to process the right subarray iteratively. The TRE-QUICKSORT effectively performs the sort in the same manner as QUICKSORT. Therefore, TRE-QUICKSORT correctly sorts the array A.

b) The worst case scenario for stack dept occurs when the partitioning is highly unbalanced. For example, in an already sorted or reverse sorted input, the partition picks the smallest (or largest) element as the pivot. So, we get,

TRE-QUICKSORT (A, 1, n)
↳ TRE-QUICKSORT(A, 1, n-1)
↳ TRE-QUICKSORT(A, 1, n-2)
⋮
↳ TRE-QUICKSORT(A, 1, 1)

This results in n recursive calls, leading in a stack depth of $\theta(n)$.

For, an array [1, 2, 3, 4, 5], the graphical representation will be

$$T(5) \rightarrow T(4) \rightarrow T(3) \rightarrow T(2) \rightarrow T(1)$$

n=5, therefore 5 steps to complete the recursion.

This confirms that in the worst case, the stack depth of TRE-QUICKSORT will be $\theta(n)$.

④ To ensure a maximum stack depth of $\Theta(\log n)$, while maintaining the $O(n \log n)$ expected running time of the algorithm, we need to make sure to perform the recursive operation on the smaller subarray and iterate on the larger subarray.

The modified TRE-QUICKSORT will be;

```
TRE-QUICKSORT-MOD (A, p, r)
  while p < r:
      q = PARTITION (A, p, r)
      if (q-p < r-q):
          TRE-QUICKSORT-MOD (A, p, q-1)
      else:
          p = q+1
          TRE-QUICKSORT-MOD (A, q+1, r)
          r = q-1
```

This modified algorithm applies recursion on the smaller portion and iterates over the larger one in each iteration. Since, each recursive call operates on half the elements (at most), the depth follows a logarithmic pattern. The runtime will be,

$$O(n \log n) + O((\log n)^2) = O(n \log n)$$

Therefore, keeping the running time at $O(n \log n)$, the worst case stack depth for the modified algorithm will be $\Theta(\log n)$.

Answer to the Question No. - 1

To show that $\sqrt{n} = \Omega(\log_2^3 n)$ using Definition 2, we proceed as follows:

Ω-notation: $g(n) \in \Omega(f(n))$

If there exist constants $c > 0$ and $n_0 \geq 0$ such that:

$$g(n) \geq c \cdot f(n) \qquad \text{for all } n \geq n_0.$$

Here, we need to show:

$$\sqrt{n} \geq c \cdot \log_2^3 n \quad \text{for sufficiently large n (for some } n_0)$$

Let,

$$g(n) = \sqrt{n}$$

$$f(n) = \log_2^3 n = \left(\frac{\ln n}{\ln 2}\right)^3$$

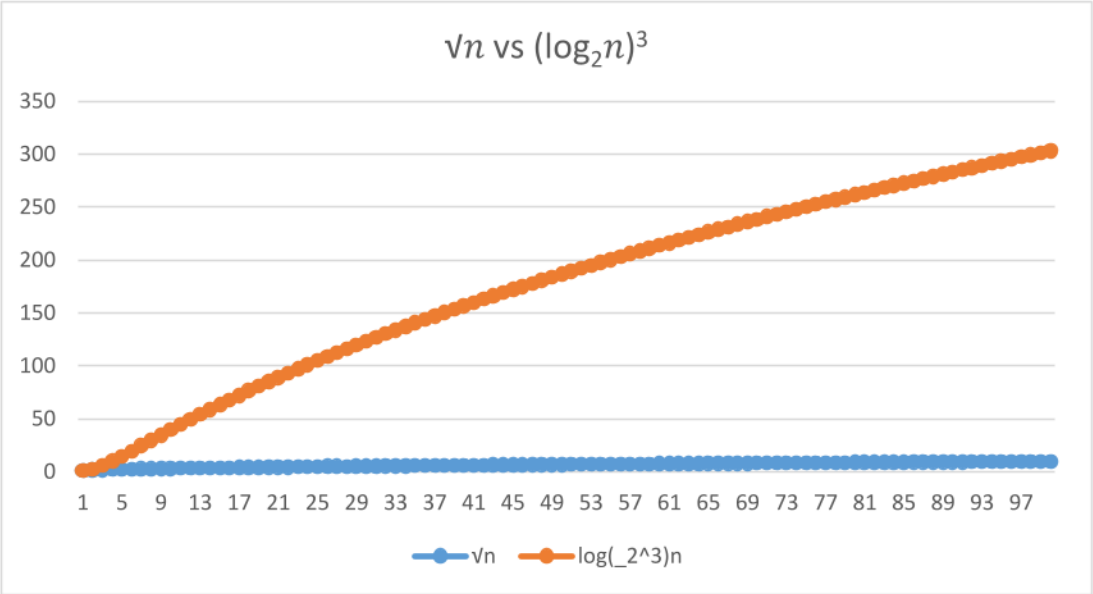
So, we need to show that:

$$\sqrt{n} \geq c \cdot \left(\frac{\ln n}{\ln 2}\right)^3$$

Dividing both sides by $\ln^3 n$, we get,

$$\frac{\sqrt{n}}{\ln^3 n} \geq c \cdot \left(\frac{1}{\ln 2}\right)^3$$

For choosing constants c and n_0,



We can let $c = (\ln 2)^3$ as a positive constant and we need to choose n_0 large enough such that the equation holds for all $n \geq n_0$.

As the right side is constant, let's analyze the left side.

Since, $n \to \infty$, the term $\sqrt{n}$ grows much faster than $\ln^3 n$. Because logarithmic functions grow much slower than any polynomial of n.

$$\Rightarrow \lim_{n\to\infty} \frac{\sqrt{n}}{\ln^3 n}$$

$$= \lim_{n\to\infty} \frac{\frac{1}{2\sqrt{n}}}{\frac{3(\ln n)^2}{n}} = \lim_{n\to\infty} \frac{\sqrt{n}}{6(\ln n)^2}$$

$$= \lim_{n\to\infty} \frac{\sqrt{n}}{24 \ln n}$$

$$= \lim_{n\to\infty} \frac{\sqrt{n}}{48} = \infty$$

The above proof shows that, $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$

The condition for Ω-notation is satisfied because,

$$\sqrt{n} \geq c \cdot \log_{\frac{3}{2}} n \quad \text{for all } n \geq n_0.$$

So, by Definition 2, it is proven that,

$$\sqrt{n} = \Omega(\log_{\frac{3}{2}} n)$$

Answer to the Question No. – 2

If we convert the given time into microseconds we get,

1 second = 10^6 microseconds

1 minute = 6×10^7 microseconds

1 hour = 3.6×10^9 microseconds

1 day = 8.64×10^{10} microseconds

1 month = 2.59×10^{12} microseconds

1 year = 3.15×10^{13} microseconds

1 century = 3.15×10^{15} microseconds

Let the time = t, then for each expression the result will be as following:

$\log_2 n \rightarrow 2^t$
$\sqrt{n} \rightarrow t^2$
$n \rightarrow t$
$n \log n \rightarrow$ approximations by calculating $n \log n \sim t$
$n^2 \rightarrow \sqrt{t}$
$n^3 \rightarrow \sqrt[3]{t}$
$2^n \rightarrow \log_2 t$
$n! \rightarrow$ approximations by calculating $n! \sim t$

The value for the comparison of running times has been shown in the following table:

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\log_2 n$	2^{10^6}	$2^{6\times10^7}$	$2^{3.6\times10^9}$	$2^{8.64\times10^{10}}$	$2^{2.59\times10^{12}}$	$2^{3.15\times10^{13}}$	$2^{3.15\times10^{15}}$
$\sqrt{n}$	10^{12}	3.6×10^{15}	1.3×10^{19}	7.46×10^{21}	6.72×10^{24}	9.95×10^{26}	9.95×10^{30}
n	10^6	6×10^7	3.6×10^9	8.64×10^{10}	2.59×10^{12}	3.15×10^{13}	3.15×10^{15}
$n \log n$	6.24×10^4	2.8×10^6	1.33×10^8	2.76×10^9	7.19×10^{10}	7.98×10^{11}	6.86×10^{13}
n^2	1000	7745	60000	293938	1.6×10^6	5.62×10^6	5.62×10^7
n^3	100	391	1532	4420	13736	31593	146645
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

Answer to the Question No. – 3

Assuming that $k \geq 1, \epsilon > 0, and\ c > 1$ are constants, we need to find the relative asymptotic growths for each pair of expressions (A, B).

$\lg^k n$ vs n^ϵ:

$$\lim_{n\to\infty} \frac{\log^k n}{n^\epsilon}$$

Since n^ϵ grows much faster than any power of $\log n$, the limit goes to 0. We get,

$$\lim_{n\to\infty} \frac{\log^k n}{n^\epsilon} = 0$$

So, it will be **yes** for O, o but **no** for Ω, ω, θ.

n^k vs c^n:

$$\lim_{n\to\infty} \frac{n^k}{c^n}$$

Using logarithms,

$$= \lim_{n\to\infty} \frac{k \log n}{n \log c}$$

$$= \lim_{n\to\infty} e^{k \log n - n \log c}$$

Since, $n \log c$ grows faster than $k \log n$,

$$= \lim_{n\to\infty} e^{-\infty} = 0$$

So, it will be **yes** for O, o but **no** for Ω, ω, θ.

$\sqrt{n}$ vs $n^{\sin n}$:

The value for $\sin n$ is in the range between -1 to +1 and it doesn't have a consistent growth pattern. As there is no consistency, none of the asymptotic relations can be justified. So, it will be **no** for each one.

2^n vs $2^{n/2}$:

$$\lim_{n\to\infty} \frac{2^n}{2^{n/2}}$$

If we take the ratio, it would be $2^{n/2}$, which tends to ∞. It refers to the lower bound. So, it will be **yes** for Ω, ω but **no** for O, o and θ.

$n^{\lg c}$vs $c^{\lg n}$:

We can rewrite the expressions as,

$$n^{\lg c} = e^{\log c \cdot \log n}$$

$$c^{\lg n} = e^{\log n \cdot \log c}$$

Therefore,

$$n^{\lg c} = c^{\lg n}$$

As both the expressions are equal, they will grow at the same rate. So, it will be **yes** for O, Ω and θ but **no** for o and ω.

$\lg(n!)$ vs $\lg(n^n)$:

Both the functions represent the same output. As the difference is almost non-existent, they grow at the same rate. So, for this case, it will be **yes** for O, Ω and θ but **no** for o and ω.

Considering all the cases, the final results have been shown in the following table:

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

Overview

- MOTIVATION
- ALGO DEF.
- ANALYSIS - Seq. Search
- TIME
  WORK DONE
  WORST, AV CASE

  SPACE
- OPTIMALITY
- ORDER NOTATION
- Design - Bin Search.

- MOTIVATION

  — Algorithms: How to do things on a computer?

  — Other application areas study specialized algorithms.

    e.g.

    Operating Systems
    Compiler Design
    Artificial Intelligence
    Data Base Management Systems

  — Algorithm Design Techniques
  Divide and conquer, greedy, Dynamic Programming etc.

  — Basic Algorithms
  sorting, graph algo, matrix multiplication, NP-Complete problems, parallel algos.

- HISTORY
  Phrase Algorithm; Persian Author

  — Abu Jafar Mohammed ibn Musa *al Khowarizmi* (825 AD)

  — wrote a Math textbook

  — al Khowarizmi: from the town of Khowarazm (now Khiva, Uzbekistan)

- ALGORITHM DEFINITION
  An algorithm is composed of a finite number of steps, each of which may require one or more operations.

  — each OPERATION executable on a computer.

  — compute 5/0 is not WELL DEFINED.

  — algorithms must TERMINATE after a finite number of operations.

  — PROCEDURE: a nonterminating algorithm.
  e.g. OS.

- GOAL - Distinguish Algorithms.

- TIME - Computer Dependent.
  AMOUNT OF WORK - COST

  — proportional to the number of **basic operations**.

  — computer independent.

  e.g. matrix multiplication
  basic operation - multiplication & addition.

Example Problem: Sequential Search

Problem: Given an array $L[1..n]$, containing n DISTINCT entries, find the index of x, if $x \in L$, else return 0.

Input: array L, array size n, search item x.

Output: If $x \in L$ then index of x in L, else 0.

Algorithm: .

Abstract Level Description: Scan L left to right looking for x in L & return index.

Pseudocode

1. index $\leftarrow 1$
2. **While** index $\leq n$
3. **if** L[index] $= x$
4. return (index)
5. **else**
6. index $\leftarrow$ index + 1
7. return (0)

Work done

Steps 1. assignment to a register variable

While loop, if $x = L[k]$

2. k comparisons of register variables
3. k comparisons of x with L entry.
6. $k - 1$ increments of register variable.

Since step 3 is costliest & other steps are executed no more times, basic operation is *comparison of a memory and a register variable.*

cost $= k,$ if $x = L[k]$
cost $= n,$ if $x \notin L$

Time Complexity

Best Case time

- $k = 1, \quad x = L[1], \quad$ best-case input.
- $B(n) = 1, \quad$ as a function of input size.

Worst Case time

- $x = L(n)$ or $x \notin L$
- $W(n) = n$

Average-Case cost?

- Let $x \in L$
- Further, let x is equally likely to be in any position 1 through n.
- E_k: event that $x = L[k]$
- Probability of E_k happening

$$\begin{aligned} P(E_k) &= 1/n, \qquad 1 \le k \le n \\ t(E_k) &= \text{cost or time when } L[k] = x \\ &= k \end{aligned}$$

Thus, average cost

$$\begin{aligned} A(n) &= P(E_1)t(E_1) + P(E_2)t(E_2) + \ldots + P(E_n)t(E_n) \\ &= \sum_{k=1}^{n} P(E_k)t(E_k) \\ &= \sum_{k=1}^{n} \frac{1}{n}k \\ &= \frac{1}{n}\sum_{k=1}^{n} k \\ &= \frac{1}{n}\left(\frac{n(n+1)}{2}\right) \\ &= \frac{n+1}{2} \end{aligned}$$

Did we average over all possible inputs?

Let us allow the possibility that x might not be in L.

$$E_0 = \text{event that } x \notin L$$

Let

$$\begin{aligned} P(x \in L) &= q \\ P(x \notin L) &= 1 - q = P(E_0) \end{aligned}$$

For $k \geq 1$,

$$\begin{aligned} P(E_k) &= P(x \in L \text{ and } x = L[k]) \\ &= P(x \in L) P(x = L[k] \text{ given that } x \in L) \\ &= q \frac{1}{n} = \frac{q}{n} \end{aligned}$$

Thus,

$$\begin{aligned} A(n) &= \sum_{k=0}^{n} P(E_k) t(E_k) \\ &= \sum_{k=1}^{n} P(E_k) t(E_k) + P(E_0) t(E_0) \\ &= \sum_{k=1}^{n} \frac{q}{n} k + (1-q)n \\ &= \frac{q}{n} \sum_{k=1} k + (1-q)n \\ &= \frac{q}{n} \frac{n(n+1)}{2} + (1-q)n \\ &= q \frac{n+1}{2} + (1-q)n \end{aligned}$$

Thus,

- if $q = 1, \quad x \in L, \quad A(n) = \frac{n+1}{2}$
- if $q = 0, \quad x \notin L, \quad A(n) = n$
- if $q = 1/2,$

$$\begin{aligned} A(n) &= \frac{1}{2}\,\frac{n+1}{2} + \frac{1}{2}n \\ &= \frac{n+1+2n}{4} \\ &= \frac{3n}{4} + \frac{1}{4} \end{aligned}$$

Cost, Work = time complexity.

Space Complexity

Extra space used apart from the input and the program (and the output, if required by specification).

$$S(n) = 1$$
(1 register variable)

Trade-off between time & space

If number are between 1 and 100, then a prepared index array $A[1 \ldots 100]$ such that

$A[i] = j$, if $L[j] = i$; else $A[i] = 0$

can yield $W(n) = O(1)$.

(*characteristic array*)

Optimality

- Is the sequential search algo. optimal?
- Is there another algo. which solves the same problem using fewer number of comparisons?
- Theorem: In the worst case,

$$W(n) = n$$

- Proof by contradiction

  — If there is another algo B whose $W(n) < n$, then the element not seen by B may be x & B would be incorrect.

  ($O(\log n)$ on an ordered list & $O(1)$ in a characteristic vector alternatives only when performing multiple searches.)

  — **Lower Bound** on the number of comparisons for searching in an unordered list is n.

Problem Complexity

Def. (Worst case) Complexity of a problem is the min number of operations needed to solve the problem in the worst case using particular resources.

eg. Matrix Multiplication
$C_{nn} = A_{\text{n x n}} * B_{\text{n x n}}$
$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + \cdots + A_{in}B_{nj}$

for $i \leftarrow 1$ to n
  for $j \leftarrow 1$ to n
    $C_{ij} \leftarrow 0$
    for $k \leftarrow 1$ to n
      $C_{ij} = C_{ij} + A_{ik}B_{kj}$

- Basic operation: Multiplication (addition can be ignored)
- $W(n) = n^3$
- lower bound on number of multiplications $= n^2$
  lower bound on the problem complexity of matrix multiplication $= n^2$.
- best algorithm found has complexity $n^{2.367}$

Average-case problem complexity

Space complexity of a problem

- matrix multiplication $O(1)$
- sorting on comparison model $O(1)$.
- matching parentheses $O(n)$.

2 GROWTH RATES AND ASYMPTOTIC NOTATIONS

Goal:

- Categorize algorithms based on their asymptotic growth rate.
- Ignore (small) constant of proportionality & small inputs
- Estimate upper bound on growth rate of time complexity function

def1: (For arbitrary functions) $f(n) \in O(g(n))$ if there exist two positive constants c and n_0 such that

$$|f(n)| \leq c|g(n)|$$

for all $n \geq n_0$.

eg.

1. $f(n) = 5n \ g(n) = n^2$
   $5n_0 \leq cn_0^2$
   $\Rightarrow 5 \leq cn_0$
   $\Rightarrow n_0 = 5, c = 1$
   Thus, $5n = O(n^2)$
2. $f(n) = 10^6 n^2 \ g(n) = n^2$
   To prove: $(10^6 n^2) \leq cn^2$ for $n \geq n_0$
   choose $C = 10^6$
   Then $n_0 = 1$
   $\Rightarrow (10^6 n^2) = O(n^2)$

3. $A(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$

   a polynominal of degree m, $n, m \geq 0$ for all i, $a_i \in R$

   To Prove: $A(n) = O(n^m) = a_m n^m + O(n^{m-1})$

   Example: $5n^3 + 2n^2 - 5 \ = \ O(n^3) \ = \ 5n^3 + O(n^2)$

   Proof:

   To Prove: $|A(n)| \ \leq \ c|n^m|$

   $|A(n)| \leq |a_m| n^m + |a_{m-1}| n^{m-1} + \ldots + |a_1| n + |a_0|$

   $\leq n^m (|a_m| + \frac{|a_{m-1}|}{n} + \ldots + \frac{|a_1|}{n^{m-1}} + \frac{|a_0|}{n^m})$

   $\leq n^m (|a_m| + |a_{m-1}| + \ldots + |a_0|)$

   $c = |a_m| + |a_{m-1}| + \cdots + |a_0|$

   $n_0 = 1$

   Example: $5n^3 + 2n^2 - 5 \ = \ O(n^3)$
   c $= 5 + 2 + |-5| = 12$

   $5n^3 + 2n^2 - 5 \ \leq \ 12n^3$
   for all $n \geq 1$.

2.1 ORDER NOTATION FOR POSITVE FUNCTIONS

def2: Let

$$f : N \to R^*$$

$$g : N \to R^*$$

$$g(n) = O(f(n))$$

, ie.,

$$g(n) \in O(f(n))$$

,
if for some $c \in R^+$ and some $n_0 \in N$

$$g(n) \leq cf(n)$$

for all $n \geq n_0$, where
$R^* = R^+ \cup \{0\}$,
R^+ =set of positive reals
$N = \{0, 1, 2, 3, \ldots\}$.

def3: $f \in O(g)$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} \to c$ for some $c \in R^*$.

Examples:

1. $10^6 n^2 = O(n^2)$
   $\lim_{n\to\infty} \frac{10^6 n^2}{n^2} \to 10^6$

2. $5n \in O(n^2)$
   $\lim_{n\to\infty} \frac{5n}{n^2} \to 0$

3. Prove that $2^n \neq O(n^m)$ for any integer m.

   (By Contradiction:) Suppose $2^n = O(n^m)$

   $\lim_{n\to\infty} \frac{2^n}{n^m} = \lim_{n\to\infty} \frac{2^n \log 2}{mn^{m-1}}$

   $= \lim_{n\to\infty} \frac{(\log 2)^2\ 2^n}{m(m-1)n^{m-2}}$

   $= \lim_{n\to\infty} \frac{(\log 2)^m 2^n}{m(m-1)\cdots(2)1} \to \infty$

   Thus, polynomial algorithms are distinctly superior to exponential-time algorithms.

(Project should have polynomial algo's only)

2.2 Θ AND OTHER NOTATIONS

def: If $f(n) = O(g(n))$
then $g(n) = \Omega(f(n))$.

Thus, $O(f)$ contains all function with growth rate "equal or slower."

$\Omega(f)$ contains all function with growth rate "equal or faster."

def: If $f(n) = O(g(n))$ and $g(n) = O(f(n))$
then $f(n) = \Theta(g(n))$.

Equivalently, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in R^+$.

Thus,

$$\Theta(f)$$

$= O(f) \cap \Omega(f)$

In terms of growth rate,
$\Theta(f)$ contains functions growing at the same rate as f.

$O(f)$ contains function growing no faster than f

$\Omega(f)$ contains function at least as fast as f

def: (Small o and ω notations):

$f = o(g)$ if $\lim_{n\to\infty} f/g \to 0$

$f = \omega(g)$ if and only if $g \in o(f)$

2.3 $\sqrt{(}n)$ VERSUS $\log_2^3 n$

show $\sqrt{n} = \Omega(\log_2^3 n)$

$\Rightarrow \ln^3 n = O(\sqrt{n})$

$\frac{\ln^3 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{3\log^2 n 1/n}{1/2n^{-1/2}}$

$= \lim_{n\to\infty} 6\frac{\log^2 n}{\sqrt{n}} = \lim_{n\to\infty} 6\frac{2\log n 1/n}{1/2n^{-1/2}}$

$= \lim_{n\to\infty} 24\frac{\log n}{\sqrt{n}} = \lim_{n\to\infty} 24\frac{1/n}{1/2n^{-1/2}}$

$= \lim_{n\to\infty} \frac{48}{\sqrt{n}} \to 0$

$\Rightarrow \log^3 n = O(\sqrt{n})$

3 H.W.1

- Use defn. 2 to show that $\sqrt{n} = \Omega(\log_2^3 n)$

- Q. 1-1, p. 13
- Q. 3-2, p. 58

2 SEARCHING ON A SORTED LIST

Problem: Given a list $L[1, \ldots, n]$ containing

keys such that $L[i] \leq L[i+1]$, for $i = 1, 2, \ldots, n-1$.

Problem is to find out if a given x is in L.

I. Use sorted property on sequential search

Quit search as soon as x is less than $L[i]$ and declare that $x \notin L$.

Could be shown that

$$A(n) = O(n) \approx n/2$$

$W(n)$ is still $O(n)$.

2.1 Jump Search

2.1.1 Jump Search (Divide and Conquer:)

Scan every ith entry of L for a fixed i. Suppose you have established that

$$L[2i] < x < L[3i]$$

then sequentially search $L[2i+1], L[2i+2], \ldots, L[3i-1]$.

$$W(n) = \frac{n}{i} + (i-1)$$

because there are $\frac{n}{i}$ partitions, and there are $(i-1)$ items to search sequentially within a partition.

eg. $i = 4, w(n) = n/4 + 3$

$i = 10, w(n) = n/10 + 9$

$i = 100, w(n) = n/100 + 99$

but still $O(n)$ for fixed i.

How about i depending n?

eg. $i = \log n$

$$\Rightarrow w(n) = \frac{n}{\log n} + \log n - 1 = O(\frac{n}{\log n})$$

better than $O(n)$ of seq search.

($n \notin O(\frac{n}{\log n})$)

eg. $i = \sqrt{n}$

$W(n) = n/\sqrt{n} + \sqrt{n} - 1$

$= 2\sqrt{n} - 1 = O(\sqrt{n})$

better than $O(\frac{n}{\log n})$.

eg. $i = \frac{n}{\log n}$

$W(n) = \frac{n}{\frac{n}{\log n}} + \frac{n}{\log n} - 1$

$= \log n + \frac{n}{\log n} - 1 = O(\frac{n}{\log n})$

Let us minimize for i,

$W(n) = n/i + i - 1$

$\frac{d}{di}(W(n)) = -\frac{n}{i^2} + 1$

set to 0 and solve, gives $i = \sqrt{n}$.

Hence $W(n) = O(\sqrt{n})$ is the best possible with the approach.

2.1.2 Recursively apply partitioning in a smaller list

Let us partition into k intervals of size n/k each

$$W(n) = n/i + i - 1 \tag{1}$$

$$W(n) = \frac{n}{n/k} + n/k - 1 \tag{2}$$

$$= k + \frac{n}{k} - 1 \tag{3}$$

$$= O(n/k) \tag{4}$$

$$\tag{5}$$

If we apply partitioning recursively in the sublist, we get

$$W(n) = k + W(n/k)$$

for k partitions and each partition of size n/k.

Since, in order to identify a partition, we need not compare x with $L[1]$,

$$W(n) = k - 1 + W(n/k)$$

Say $k = 4$, then

$$\begin{aligned}
W(n) &= 3 + W(n/4) \\
&= 3 + \left(3 + W\left(\frac{n/4}{4}\right)\right) \\
&= 3 + 3 + W\left(\frac{n}{4^2}\right) \\
&= 3 + 3 + \left(3 + W\left(\frac{n}{4^3}\right)\right) \\
&= 3 * 3 + W\left(\frac{n}{4^3}\right) \\
&= 4 * 3 + W\left(\frac{n}{4^4}\right) \\
&\ldots \\
&= i * 3 + W\left(\frac{n}{4^i}\right)
\end{aligned}$$

How large can i be?

Eventually, partition size will become equal to 1, when

$n = 4^i$ or $\log_4 n = i$

Then $W(1) = 1$

Thus,

$W(n) = 3 \log_4 n + W(1)$

$$W(n) = 3 \log_4 n + 1$$

$= O(\log_4 n)$

– much better than $\sqrt{n}$.

In general, for any fixed k,

$$w(n) = (k-1) \log_k n + 1$$

2.2 RECURSIVE JUMP SEARCH: With Best Value for k

Minimize $W(n)$ w.r.t k.

Find $\frac{dw}{dk}$ & solve by setting to 0.

$$\begin{aligned} w(n) &= (k-1)\log_k n + 1 \\ &= (k-1)\frac{\log_e n}{\log_e k} + 1 \\ \frac{dw(n)}{dk} &= (k-1)\log_e n(-1)(\log_e k)^{-2}\frac{1}{k} + \frac{\log_e n}{\log_e k} \\ &= -\frac{k-1}{k}\frac{\log_e n}{(\log_e k)^2} + \frac{\log_e n}{\log_e k} \end{aligned}$$

Set $\frac{dw}{dk} = 0$ and solve to get

$\frac{k-1}{k} = \log_e k$

$\Rightarrow \log_e k = 1 - \frac{1}{k} < 1$

$\Rightarrow k < e^1 = e = 2.7$

$\Rightarrow k = 2$ (can not be 1)

Further check that $\frac{d^2w}{dk^2}$ at $k = 2$ is ≥ 0 for a minimum value.

Thus, $k = 2$ is the best. So, dividing in 3 partition is not better than that in 2.

$w(n) = k - 1 + w(n/k)$

$= 2 - 1 + w(n/2)$

$= \log_2 n + 1$

$$w(n) = \lfloor \log_2 n \rfloor + 1$$

Binary Search: For binary search, we assumed that n is a power of 2.

If not, $w(n) = \lfloor \log_2 n \rfloor + 1$

$A(n) = \log_2 n + 1/2$

for binary search.

3 Optimality of Binary Search

Computation Model: Only operation allowed is comparison: Comparison Model

To Show: Binary Search is optimal in the class of search algorithms on an ordered list that can perform no other operation on the entries except comparison.

3.1 Descision Trees

- **Sequential Search**

  $n = 16$

  $w(n) = n$

- **Jump Search**

  $n = 16$ sublist size $= \sqrt{16} = 4$

  $w(16) = 7 = 2n - 1 = 2 \cdot 4 - 1 = 7$

- **Binary Search**

  Middle= $\lfloor \frac{first+last}{2} \rfloor$

  $w(16) = 5 = 4 + 1 = \lfloor \log_2 + 1 \rfloor$

Proof: (Binary search is optimal)

- Numbers of nodes in any decision tree is $\geq n$

- Minimum numbers of levels in any binary tree with n nodes is $\geq \lfloor \log_2 n \rfloor + 1$

  (H.W.)

- $\Rightarrow 1 + \lfloor \log_2 n \rfloor$ is a lower bound on problem complexity

- $\Rightarrow$ Binary Search is optimal

2 SORTING BY RANKING

{**rank** of an item=number of items smaller}

rank each item by counting;

then place each item according to its rank.

If duplicate, then place it at the nearest slot to the right.

$W(n) = O(n(n-1)) = A(n) = B(n)$

$S(n) = O(n)$

To handle duplicates, redefine

rank(i) = number of items smaller than i or, if equal, occurring before i.

Oblivious Algorithm

3 SORTING BY SWAPPING

Bubble Sort–good when input is nearly sorted

$W(n) = O(n^2/2)$

$S(n) = O(1)$

Odd-Even Exchange Sort

a) odd-even compare & exchange

b) even-odd compare & exchange

c) repeat step (a) and (b) if exchange-count > 0

$W(n) = O(n^2)$

$S(n) = O(1)$

4 SORTING BY INSERTION

Online algorithms.

Linear Insertion Sort

Insert the next element in the ordered list prepared so far by sequential search & shifting.

$W(n) = O(n^2/2)$

$S(n) = O(1)$

$O(n)$ time on a sorted list

Binary Insertion sort

perform binary search to find location for insertion.

$W(n) = O(n \log n) + O(n^2)$.

Tree Sort

Insert into a binary search tree, then traverse tree in-order.

$W(n) = O(n^2)$

$S(n) = O(n)$

$A(n) = B(n) = O(n \log n) + O(n)$

5 SORTING BY SELECTION

Offline algorithms

Selection sort

find maximum & replace with the concurrent last

$W(n) = O(n^2)$

$S(n) = O(1)$

$O(n^2)$ even on a sorted list

Tournament Sort

$W(n) = A(n) = B(n) = O(n \log n)$

$S(n) = O(n)$ for the tournament tree

Heap Sort

- construct a max heap -$O(n)$
- delete root and update heap repeatedly -$O(n \log n)$

$W(n) = A(n) = O(n \log n)$

$S(n) = O(1)$

5.1 Heap Sort

- Restore-Heap(i): $O(h)$ where h is the height of the node i.

- Construct Heap:

  For $i = \left\lfloor \frac{n}{2} \right\rfloor$ down to 1 Restore-Heap(i)

  $O(n)$

- Heap Sort:

  1. Construct Heap – $O(n)$

  2. For i:=n down to 2 – $O(n \log n)$

     exchange $L[1]$ with $L[i]$

     decrement heap size

     Restore-Heap(1)

- Time Complexity of deletion phase in Heap Sort

$$\begin{aligned} W(n) &= 2\log n + W(n-1) \\ &= 2\sum_{i=1}^{n} \log i \\ &\leq 2\int_{1}^{n+1} \log x dx \end{aligned}$$

$$= [2(x \log x - x)]_1^{n+1}$$

$$= 2n \log n - 2n$$

5.1.1 Heapsort Construction

Construct Heap:

for $i = \lfloor n/2 \rfloor$ downto 1

Restore-heap(i)

Time Complexity of Iterative Algorithm for Heap Construction:

$$\sum_{h=1}^{\lfloor \log n \rfloor} \lceil n/2^{h+1} \rceil O(h)$$

$$= O\left(n \sum_{h=1}^{\log n} (h/2^h)\right)$$

$= O(n)$ (pp. 159)

Consider $\sum_{h=1}^{\log n} h/2^h$

Let

$$x = 1/2 + 2/2^2 + 3/2^3 + 4/2^4 + \cdots + y/2^y \quad (1)$$

$$2x = 1 + 2/2^1 + 3/2^2 + 4/2^3 + \cdots + y/2^{y-1} \quad (2)$$

$$x = 1 + 1/2 + 1/2^2 + \cdots + 1/2^{y-1} \quad (3)$$

$$-y/2^y \quad (4)$$

$$= \frac{(1/2)^y - 1}{1/2 - 1} - y/2^y \quad (5)$$

$$= 2(1 - (1/2)^y) - y/2^y \quad (6)$$

$$\leq 2 \quad (7)$$

5.1.2 Heapsort: Recursive Construction

Construct-Heap(n):

construct left subheap

construct right subheap

Restore-heap(1)

Time Complexity:

$$
\begin{aligned}
W(n) &= 2w(n/2) + \log n && (9)\\
&= \log n + 2w(n/2) && (10)\\
&= \log n + 2\left(\log(n/2) + 2w(\frac{n/2}{2})\right) && (11)\\
&= \log n + 2\log n - 2\log 2 + && (12)\\
&\quad 2^2 w(n/2^2) && (13)\\
&= \log n + 2\log n - 2\log 2 + && (14)\\
&\quad 2^2\left(\log(\frac{n}{2^2}) + 2w\left(\frac{n/2^2}{2}\right)\right) && (15)\\
&= \log n + 2\log n - 2\log 2 + && (16)\\
&\quad +2^2\log n - 2^2\log(2^2) + 2^3 w(n/2^3) && (17)\\
&\quad \cdots && (18)\\
&= \log n + 2\log n - 2\log 2 + && (19)\\
&\quad +2^2\log n - 2^2\log(2^2) + \cdots + && (20)\\
&\quad 2^k\log n - 2^k\log(2^k) + 2^{k+1}w(n/2^{k+1}) && (21)\\
&= \log n(1 + 2 + 2^2 + \cdots + 2^k) && (22)\\
&\quad -(2 + 2\cdot 2^2 + 3\cdot 2^3 + \cdots + k\cdot 2^k) && (23)
\end{aligned}
$$

Here, we assume that $n = 2^k$ so $k = \log n$.

Let $s = 2^0 + 2^1 + 2^2 + \cdots + 2^k$
$= \frac{2^{k+1}-1}{2-1} = 2^{k+1} - 1$

To derive this formula, one can follow these steps:

$$\begin{aligned} s &= 2^0 + 2^1 + 2^2 + \cdots + 2^k && (24)\\ 2s &= 2^1 + 2^2 + \cdots + 2^k + 2^{k+1} && (25)\\ s &= -1 + 2^{k+1} && (26) \end{aligned}$$

Thus, $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

Likewise, let

$$\begin{aligned} T &= 1 \cdot 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + \cdots + k \cdot 2^k && (27)\\ 2T &= 1 \cdot 2^2 + 2 \cdot 2^3 + \cdots + && (28)\\ &\quad (k-1) \cdot 2^k + k2^{k+1} && (29)\\ T &= -2^1 - 2^2 - 2^3 - \cdots - 2^k + && (30)\\ &\quad k2^{k+1} && (31)\\ &= k2^{k+1} - (2^1 + 2^2 + \cdots + 2^k) && (32)\\ &= k2^{k+1} - (2^{k+1} - 2) && (33)\\ &= (k-1)2^{k+1} + 2 && (34)\\ & && (35) \end{aligned}$$

$\Rightarrow W(n) = \log n(2^{K+1} - 1) - ((k-1)2^{k+1} + 2)$
$= 2n \log n - \log n - 2n \log n + 2n - 2$
$= 2n - \log n - 2$
$= O(n)$

5.1.3 Priority Queue employing Heap

Read pp. 162 (Section 6.5)

Delete Max/Min

Insert

See Exercise 6-1, pp. 166

6 SORTING BY MERGING

- **Mergesort**

  — $W(n) = O(n) + 2w(n/2)$
  $= n + 2w(n/2)$
  $= n + 2(n/2 + 2w(n/2^2)$
  $= n + n + 2^2 w(n/2^2)$
  $= 2n + 2^2 w(n/2^2)$
  $= 2n + 2^2(n/2^2 + 2w(n/2^3))$
  $= 3n + 2^3 w(n/2^3)$
  $\ldots$
  $= kn + 2^k w(n/2^k)$
  $w(n/2^k) = w(1) = 0$
  $n/2^k = 1$, $k = \log_2 n$
  $\Rightarrow W(n) = \theta(n \log n)$

  — Recursion tree for $W(n)$

  — $S(n) = O(n)$
  Can be reduced to $O(1)$ but algorithm slows down.
  A temporary array of half the size is required, however.

7 SORTING BY SPLITING

$W(n) = O(n^2)$ but $A(n) = O(n \log n)$

Quicksort (A, p, r):

if $p < r$ then $q \leftarrow$ Partition(A, p, r)

Quicksort(A, p, q)

Quicksort$(A, q + 1, r)$

7.1 Quicksort: Partitioning

Partition I.

$x := A[p]$; $i := p - 1$; $j := r + 1$

while (TRUE) **do**

  Repeat Decrement j **until** $A[j] \leq x$

  Repeat Increment i **until** $A[i] \geq x$

  if $i < j$

    then exchange $A[i]$ and $A[j]$

    else return j

endwhile

$x = 15$

15 7 23 5 20 3

Why do i and j never get out of array bounds?

$W(n) = n + 2$ comparisons

$W(n) = O(n^2) = O(n) + W(n-1)$

$B(n) = O(n) + 2B(n/2) = O(n \log n)$

Balance Partitioning:

- Even if each split is 1% on one side and 99% on the other, recursion tree remains logarithmic.

- Alternate good and bad splits:

Quicksort Improvements

– random x

– median of first, middle and last items

– insertion sort for $n \leq 15$

$S(n) = O(n)$
(See 7-4; reduces $S(n)$ to $O(\log n)$)

7.2 Quicksort: Partitioning II

Input: A[p..r]

Invariants: {All items in $A[2..i]$ are $<$ the pivot.}

{All items in $A[(i+1)..(unknown-1)]$ are $\geq$ the pivot}

$x = A[p];\ i := p;$

for $unknown := p+1$ **to** r **do**

  if $A[unknown] < x$ **then**

    $i := i+1$; swap$(A[i], A[unknown])$

swap$(A[1], A[i])$

$W(n) = n-1$ comparisons

15 7 23 5 20 3

23 _____

7.3 Av. Case Complexity of Quicksort

Assume

- all keys distinct
- all permutations equally likely

Probability that split point is i, for $1 \leq i \leq n$, is $1/n$

$$A(n) = (n-1) + \frac{1}{n}(A(0) + A(n-1) +$$

$$A(1) + A(n-2) + \cdots A(n-1) + A(0))$$

$$= n - 1 + \frac{2}{n}\left(A(0) + A(1) + \cdots + A(n-1)\right)$$

$$A(n) = n - 1 + \frac{2}{n}\Sigma_{i=2}^{n-1} A(i), \quad A(0) = A(1) = 0$$

$$(n)A(n) = (n)(n-1) + 2\Sigma_{i=2}^{n-1} A(i)$$

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

$$(n-1)A(n-1) = (n-1)(n-2) + 2\Sigma_{i=2}^{n-2} A(i)$$

$$nA(n) - (n-1)A(n-1)$$

$$= n(n-1) - (n-2)(n-1) + 2A(n-1)$$

$$nA(n) - (n+1)A(n-1) = 2(n-1)$$
$$\frac{A(n)}{n+1} - \frac{A(n-1)}{n} = \frac{2(n-1)}{n(n+1)}$$

Let $B(n) = \frac{A(n)}{n+1}$ (*Changing Variable*)

Since $A(1) = 0,\ B(1) = 0$

$$\Rightarrow B(n) - B(n-1) = \frac{2(n-1)}{n(n+1)}$$
$$B(n) = \frac{2(n-1)}{n(n+1)} + B(n-1)$$
$$= \frac{2(n-1)}{n(n+1)} + \left(\frac{2(n-2)}{(n-1)(n)} + B(n-2)\right)$$
$$= B(1) + \frac{2\cdot 1}{2\cdot 3} + \frac{2\cdot 2}{3\cdot 4} + \cdots + \frac{2(n-1)}{n(n+1)}, B(1) = 0$$
$$B(n) = \sum_{i=2}^{n} \frac{2(i-1)}{i(i+1)}$$

$f(n) = \frac{2(n-1)}{n(n+1)}$ continuous decreasing function Sum_a^b f(x) <= Int_{a-1}^b f(x)dx - pp. 51

$$B(n) \le \int_2^n f(x)dx, \quad f(1) = 0$$
$$\frac{2(x-1)}{x(x+1)} = \frac{A}{x} + \frac{B}{x+1}$$
$$= \frac{(A+B)x + A}{x(x+1)}$$
$$\Rightarrow A = -2$$
$$A + B = 2$$
$$\Rightarrow B = 4$$

$$\Rightarrow f(x) = \frac{4}{x+1} - \frac{2}{x}$$

$$\int_2^n f(x)dx = \int_2^n \left(\frac{4}{x+1} - \frac{2}{x}\right) dx$$

$$= (4\ln(x+1) - 2\ln x)\,|_2^n$$

$$= 4\ln(n+1) - 2\ln n - 4\ln 3 + 2\ln 2$$

$$\approx 2\ln n$$

$$\Rightarrow B(n) \leq 2\ln n$$

$$A(n) = (n+1)B(n) \leq 2(n+1)\ln n$$

$$\Rightarrow A(n) \leq 1.4(n+1)\log_2 n$$

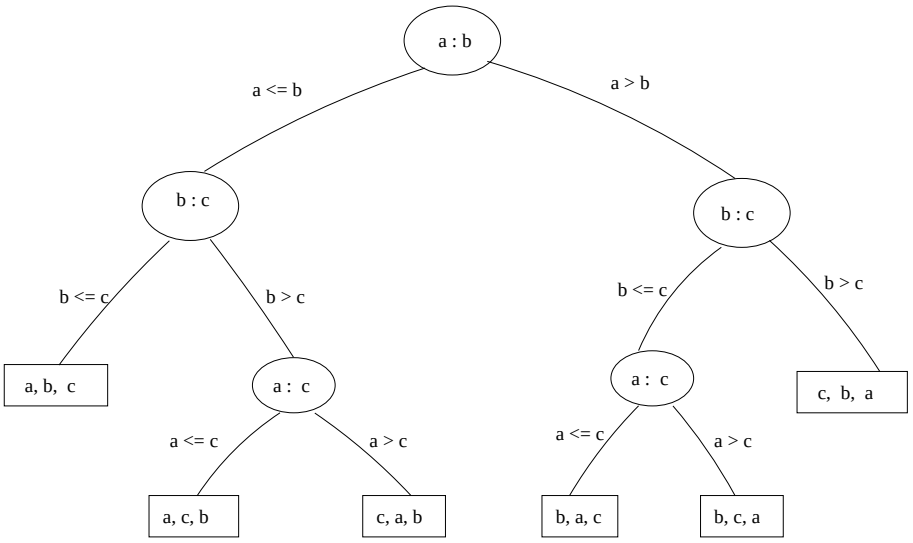
8 LOWER BOUND

(a) Local exchange only

each accomplishes in undoing one inversion
5 1 4 7 2 has inversions (5,1),(5,4),(5,2),(4,2),(7,2)
(n n-1 ... 2 1) has $n(n-1)/2$ inversions
$\Rightarrow O(n^2/2)$ lower bound

(b) lower bound on comparison based sorting
example: a b c - there are 3! outputs.
(for n numbers there are n! outputs)
Every decision tree to sort n numbers must have atleast n! leaf nodes
Every decision tree to sort n numbers must have atleast 2n! - 1 nodes
Every decision tree to sort n numbers must have depth atleast $\log_2 n!$ leaf nodes
Depth is the lower bound worst case time complexity for this class of algorithms
$\log_2 n! = log_2(n * (n-1) * (n-2) * (n-3).....1)$
$\log_2 n! = log_2(n) + log_2(n-1) + log_2(n-2) + log_2(n-3)..... + log_2(1)$
$\log_2 n! = \sum_{j=1}^{n} \log_2 j$
$\leq \int_1^n \log_2 x dx$
$= (x \log x - x)|_1^n$
$= n \log n - n$

Lower Bound on Comparison-Based Sorting Algorithm
Decision Tree for 3 numbers



Depth of binary tree with $n!$ leaves
$\geq \log_2 n!$
$\geq \int_1^n \log x dx$
$\geq n \log n - n$

9 SHELL SORT (Donald Shell)

Sort subarray comprising every h_i location, for a few selected hop sizes h_i, $k \geq i \geq 1$, and final $h_1 = 1$
$h_1 = 1$ always use insertion sort for sorting

with $h_2 = 1.72n^{1/3}$

$W(n) = (\frac{n}{1.72n^{1/3}})^2 + 1.72n^{1/3} + n^2$
$= \frac{n^2}{1.72n^{1/3}} + n^2$
$= \frac{n^{5/3}}{1.72} + n^2$
$= O(n^2)$

for $h_k = 2^k - 1$, $1 \leq k \leq \lfloor \log n \rfloor$
$W(n) = O(n$

$2^5 - 1 = 31 = (11111)_2$
$2^4 - 1 = 15 = (1111)_2$

for h_k is an integer of the form $2^i 3^j, h_k < n$
$W(n) = O(n(\log n)^2)$

$2^0 3^0, 2^1 3^0, 2^0 3^1, 2^1 3^1, 2^2 3^1$
1 2 3 6 12
too many h'_ks, hence overhead is large.