For a given function $g(n)$, the set little-omega: $(g(n))= \{f(n): \forall c> 0,\exists\ n0> 0$ such that $\forall n >= n_0$, we have $0 <= cg(n) < f(n)\ \}$.
$F(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity:
$lim\ [f(n) / g(n)] = \alpha$.
$n\text{->}\alpha$
$g(n)$ is a *lower bound* for $f(n)$ that is not asymptotically tight.

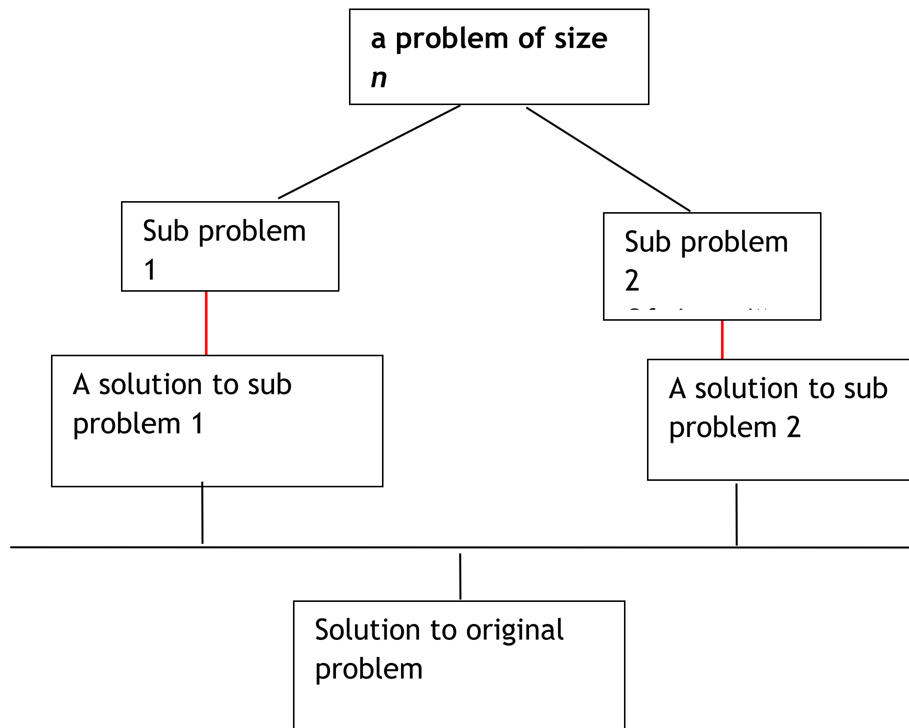
....

Chapter-2
Divide and Conquer

2.1 General Method

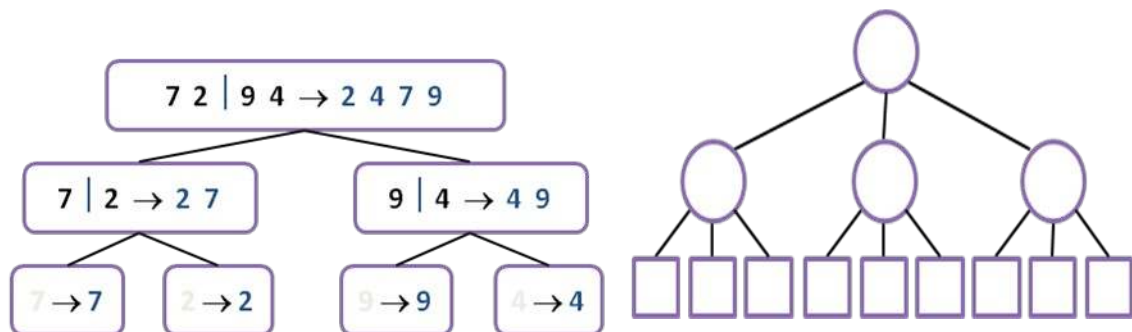
Definition

Divide the problem into a number of sub problems; conquer the sub problems by solving them recursively. If the sub problem sizes are small enough, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem.



Divide-and conquer is a general algorithm design paradigm

Divide: divide the input data S in two or more disjoint subsets S_1, S_2,
Recursively: solve the sub problems recursively
Conquer: combine the solutions for S_1, $S_{2...}$ into a solution for S
The base case for the recursion is sub problems of constant size. Analysis can be done using recurrence equations



Algorithm

Algorithm D-and-C (n: input size)
{
if n ≤ n0 /* small size problem*/
Solve problem without further sub-division;
Else
{
Divide into m sub-problems;
Conquer the sub-problems by solving them
Independently and recursively; /* D-and-C(n/k) */
Combine the solutions;
}
}

Advantage

Straight forward and running times are often easily determined

2.2 Divide-and-Conquer Recurrence Relations

Suppose that a recursive algorithm divides a problem of size n into parts, where each sub-problem is of size n/b. Also, suppose that a total number of g(n) extra operations are needed in the conquer step of the algorithm to combine the solutions of the sub-problems into a solution of the original problem. Let f (n) is the number of operations required to solve the problem of size n. Then f satisfies the recurrence relation and it is called divide-and-conquer recurrence relation.

$$F(n) = a\, f(n/b) + g(n)$$

The computing time of Divide and conquer is described by recurrence relation.

T (n) = {g(n) where n small
{T (n1) +T (n2) +………. + T (n_k) + f (n) other wise

T (n) is the time for Divide and Conquer on any input of size n and g (n) is the time to compute the answer directly for small inputs. The function of f (n) is the time for dividing P combining solutions to sub problems. For divide-and-conquer-based algorithms that produce sub problems of the same type as the original problem, then such algorithm described using recursion.

The complexity of many divide-and-conquer algorithms is given by recurrence of the form.

T (n) = {T (1) n=1

{a T(n/b) + f(n) n>1 where a and b are known constants, and n is a power of b ($n=b^k$). One of the methods for solving any such recurrence relation is called substitution method.

Examples

If a=2 and b=2. Let T (1) =2 and f (n)=n. Than
T (n) = 2T (n/2) +n
=2[2 T (n/4) + n/2] +n
=4 T (n/4) +2n
=4[2 T (n/8) +n/4] + 2n
=8 T (n/8) + 3n
.
.
.

In general, $T(n) = 2^i T(n/2^i) + in$, for any $\log_2 n \geq i \geq 1$. In Particular, then $T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + n \log_2 n$ corresponding to choice of
$i = \log_2 n$. Thus, $T(n) = n T(1) + n \log_2 n = n \log_2 n + 2n$.

2.3 Divide and Conquer Applications

2.3.1 Min and Max
The minimum of a set of elements: The first order statistic i = 1
The maximum of a set of elements: The nth order statistic i = n
The median is the "halfway point" of the set I = (n+1)/2, is unique
When n is odd
$i = \lfloor (n+1)/2 \rfloor = n/2$ (lower median) and $\lceil (n+1)/2 \rceil = n/2+1$ (upper median), when n is even
Finding Minimum or Maximum
Alg: MINIMUM (A, n)

```
min ← A[1]
for i ← 2 to n
  do if min > A[i]
    then min ← A[i]
return min
```

How many comparisons are needed?
n – 1: each element, except the minimum, must be compared to a smaller element at least once. The same number of comparisons is needed to find the maximum. The algorithm is optimal with respect to the number of comparisons performed.

Simultaneous Min, Max

Find min and max independently
Use n – 1 comparisons for each ⇒total of 2n – 2

At most 3n/2 comparisons are needed. Process elements in pairs. Maintain the minimum and maximum of elements seen so far. Don't compare each element to the minimum and maximum separately. Compare the elements of a pair to each other. Compare the larger element to the maximum so far, and compare the smaller element to the minimum so far. This leads to only 3 comparisons for every 2 elements.

Analysis of Simultaneous Min, Max

Setting up initial values:
n is odd: compare the first two elements, assign the smallest one to min and the largest one to max
n is even:
Total number of comparisons:

n is odd: we do 3(n-1)/2 comparisons
n is even: we do 1 initial comparison + 3(n-2)/2 more comparisons = 3n/2 - 2 comparisons

Example

1. n = 5 (odd), array A = {2, 7, 1, 3, 4}
    1. Set min = max = 2
    2. Compare elements in pairs:

1 < 7 ⇒compare 1 with min and 7 with max
⇒min = 1, max = 7 } 3-comparisions
3 < 4⇒compare 3 with min and 4 with max
⇒min = 1, max = 7 } 3-comparisions
3(n-1)/2 = 6 comparisons

2. n = 6 (even), array A = {2, 5, 3, 7, 1, 4}
    1. Compare 2 with 5: 2 < 5
    2. Set min = 2, max = 5
    3. Compare elements in pairs:

3 < 7 ⇒compare 3 with min and 7 with max
⇒min = 2, max = 7 } 3-comparisons

1 < 4 ⇒compare 1 with min and 4 with max
⇒min = 1, max = 7 } 3-comparisions
3n/2 - 2 = 7 comparisons
⇒min = 1, max = 7

2.3.2 Binary Search

The basic idea is to start with an examination of the middle element of the array. This will lead to 3 possible situations: If this matches the target K, then search can terminate successfully, by printing out the index of the element in the array. On the other hand, if K<A[middle], then search can be limited to elements to the left of A[middle]. All elements to

the right of middle can be ignored. If it turns out that K >A[middle], then further search is limited to elements to the right of A[middle]. If all elements are exhausted and the target is not found in the array, then the method returns a special value such as –1.

1st Binary Search function:

```
int BinarySearch (int A[ ], int n, int K)
{
int L=0, Mid, R= n-1;
while (L<=R)
{
Mid = (L +R)/2;
if ( K= =A[Mid] )
return Mid;
else if ( K > A[Mid] )
L = Mid + 1;
else
R = Mid – 1;
}
return –1 ;}
```

Let us now carry out an Analysis of this method to determine its time complexity. Since there are no “for” loops, we cannot use summations to express the total number of operations. Let us examine the operations for a specific case, where the number of elements in the array n is 64. When n= 64 Binary Search is called to reduce size to n=32

When n= 32 Binary Search is called to reduce size to n=16
When n= 16 Binary Search is called to reduce size to n=8
When n= 8 Binary Search is called to reduce size to n=4
When n= 4 Binary Search is called to reduce size to n=2
When n= 2 Binary Search is called to reduce size to n=1.

Thus we see that Binary Search function is called 6 times (6 elements of the array were examined) for n =64. Note that 64 = 26.Also we see that the Binary Search function is called 5 times (5 elements of the array were examined) for n = 32. Note that 32 = 25 Let us consider a more general case where n is still a power of 2. Let us say n = 2k.

Following the above argument for 64 elements, it is easily seen that after k searches, the while loop is executed k times and n reduces to size 1. Let us assume that each run of the while loop involves at most 5 operations. Thus total number of operations: 5k. The value of k can be determined from the expression $2^k = n$ .Taking log of both sides $\text{Log } 2^{k} = \log n$ Thus total number of operations = 5 log n. We conclude that the time complexity of the Binary search method is O (log n), which is much more efficient than the Linear Search method.

2nd method Binary Search function

Binary-Search (*A; p; q; x)*
1. if p > *q return -1;*
2. *r = b (p + q)=2 c*
3. if *x = A[r] return r*
4. else if *x < A[r] Binary-Search(A; p; r; x)*
5. else Binary-Search(*A; r + 1; q; x)*
[2] *The initial call is Binary-Search (A; 1; n; x).*

List [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]

4	8	19	25	34	39	45	48	66	75	89	95

List length=12

Search list

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]

4	8	19	25	34	39	45	48	66	75	89	95

mid

Search list, list[0]….list[1]

Middle element

Mid=left + right/2

4	8	19	25	34	39	45	48	66	75	89	95

Sorted list for a binary search

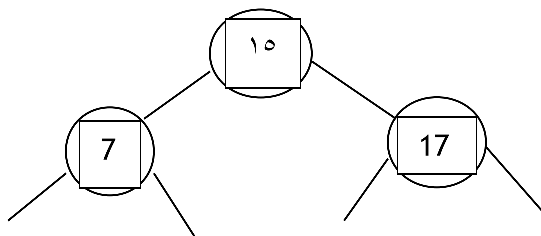
Value of first, last, mid, no of comparisons for search item 89

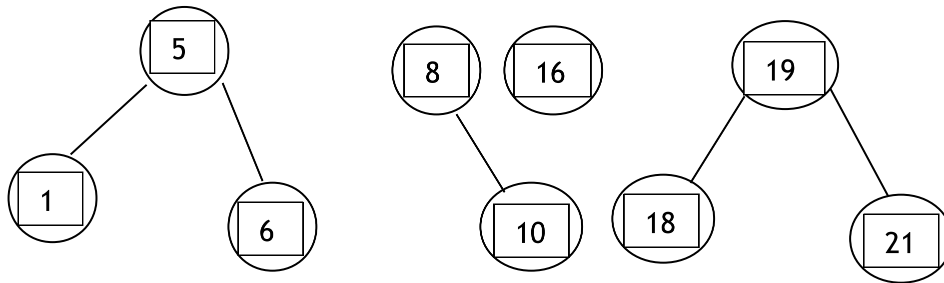
Iteration	First	Last	Mid	Last[mid]	No.of Comparisons
1	0	11	5	39	2
2	6	11	8	66	2
3	9	11	10	89	1(found it is true)

Binary search tradeoffs

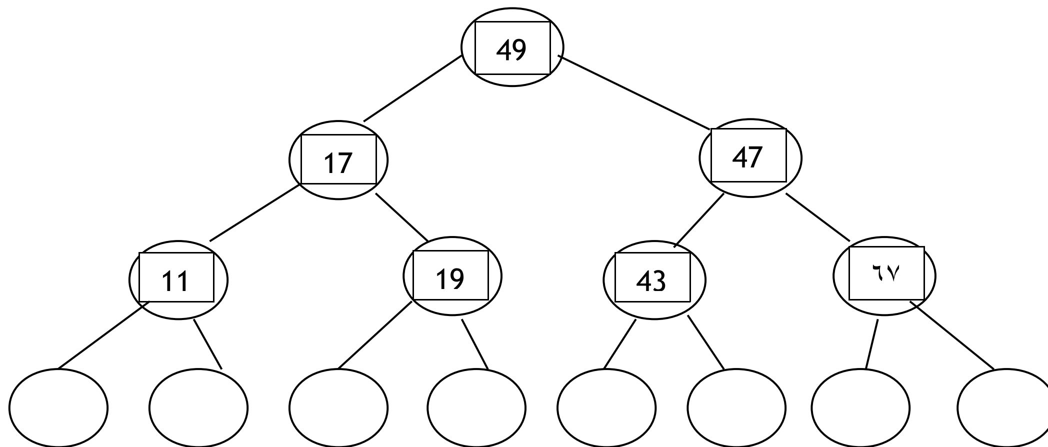
Benefit: More efficient than linear search (for array of N elements performs at most $\log_2 N$ comparisons)

Disadvantages: requires that array elements to be sorted.





Full and balanced Binary search tree



Logarithmic Time Complexity of Binary Search

Our analysis shows that binary search can be done in time proportional to the *log* of the number of items in the list this is considered *very fast* when compared to linear or polynomial algorithms .The table to the right compares the number of operations that need to be performed for algorithms of various time complexities. The computing time binary search by best, average and

Worst cases:
Successful searches
Θ (1) best, Θ (log n) average
Θ (log n) worst
Unsuccessful searches
Θ (log n) for best , average and worst case

2.3.2 Merge Sort Algorithm

Divide: Divide the n-element sequence into two subsequences of n/2 elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted sequences.

How to merge two sorted sequences:

We have two sub arrays A [p...q] and A [q+1..r] in sorted order. Merge sort algorithm merges them to form a single sorted sub array that replaces the current sub array A [p...r].

To sort the entire sequence A [1...n], make the initial call to the procedure MERGE-SORT(A,1,n).

```
MERGE-SORT (A, p, r)
{
    1.   IF p<r                       //Check for base case
    2.   THEN q=FLOOR[(p+r)/2]        //Divide step
    3.   MERGESORT (A,p,q)            //Conquer step
    4.   MERGESORT(A,q+1,r)           //Conquer step
    5.   MERGE (A, p, q, r)           //Conquer step.
}
```

The pseudo code of the MERGE procedure is as follow:

MERGE ($A, p, q,$ *and* r)

$n_1 \leftarrow q - p + 1$

$n_2 \leftarrow r - q$

Create arrays L $[1 \ldots n_1 + 1]$ and R$[1 \ldots n_2 + 1]$

FOR $i \leftarrow 1$ TO n_1

DO L$[i] \leftarrow$ A $[p + i - 1]$

FOR $j \leftarrow 1$ TO n_2

DO R$[j] \leftarrow$ A $[q + j]$

L $[n_1 + 1] \leftarrow \infty$

R $[n_2 + 1] \leftarrow \infty$

$i \leftarrow 1$

$j \leftarrow 1$

FOR $k \leftarrow p$ TO r

DO IF L $[i] \leq$ R $[j]$

THEN A $[k] \leftarrow$ L $[i]$

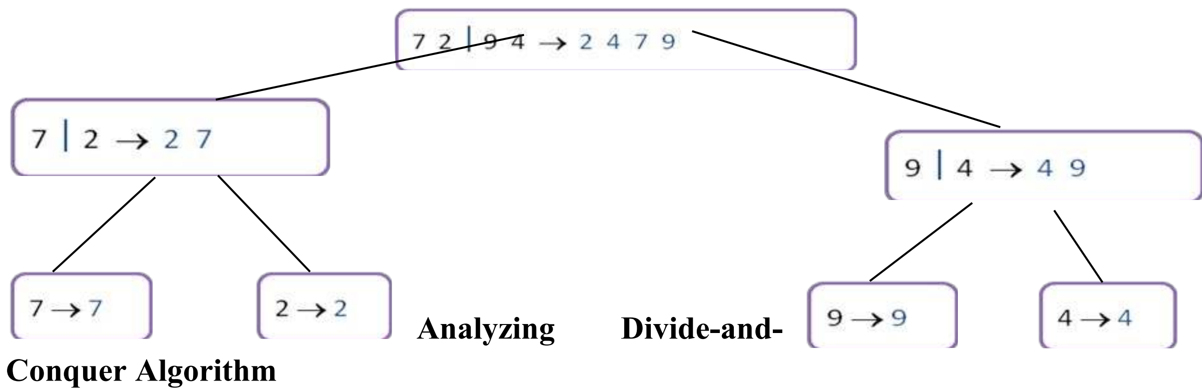
$i \leftarrow i + 1$

ELSE A[k] $\leftarrow$ R[j]

$j \leftarrow j + 1$

Merge-Sort Tree

An execution of merge-sort is depicted by a binary tree each node represents a recursive call of merge-sort and stores unsorted sequence before the execution and its partition sorted sequence at the end of the execution. The root is the initial call. The leaves are calls on subsequences of size 0 or 1.



Analyzing Divide-and-Conquer Algorithm

When an algorithm contains a recursive call to itself, its running time can be described by a recurrence equation or recurrence which describes the running time.

Analysis of Merge-Sort:

The height h of the merge-sort tree is $O(\log n)$

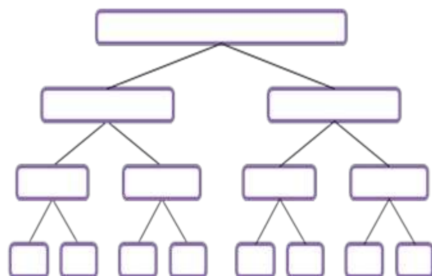
- at each recursive call we divide in half the sequence,

The overall amount or work done at the nodes of depth i is $O(n)$

- we partition and merge 2^i sequences of size $n/2^i$
- we make 2^{i+1} recursive calls

Thus, the total running time of merge-sort is $O(n \log n)$

Depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
...	…	…



Recurrence

If the problem size is small enough, say $n<=c$ for some constant c, the straightforward solution takes constant time, can be written as $\theta(1)$. If we have *a* sub problems, each of which is *1/b* the size of the original. D (n) time to divide the problem and C (n) time to combine the solution.

The recurrence is

$$T(n) = \begin{cases} \theta(1) & \text{if } n <= c \\ a\,T(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Divide: The divide step computes the middle of the sub array which takes constant time, $D(n)=\theta(1)$

Conquer: We recursively solve two sub problems, each of size n/2, which contributes 2T(n/2) to the running time.

Combine: Merge procedure takes $\theta(n)$ time on an n-element sub array. $C(n)=\theta(n)$

The recurrence is

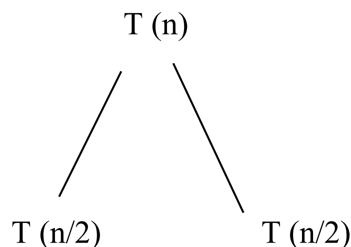
$$T(n) = \begin{cases} \theta(1) & \text{if } n=1 \\ 2T(n/2) + \theta(n) & \text{if } n>1 \end{cases}$$

Let us rewrite the recurrence

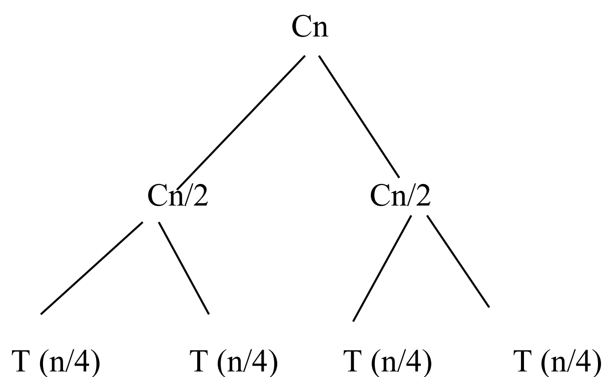
$$T(n) = \begin{cases} C & \text{if } n=1 \\ 2\,T(n/2) + cn & \text{if } n>1 \end{cases}$$

C represents the time required to solve problems of size 1

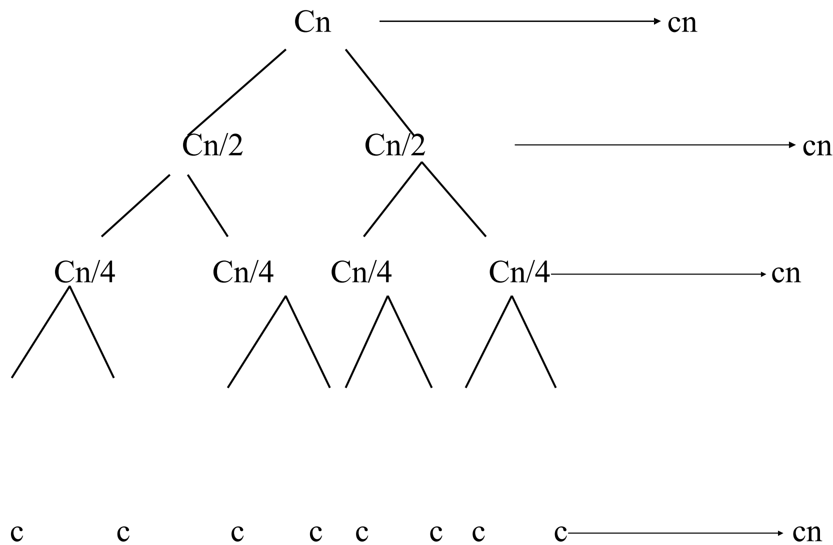
A Recursion Tree for the Recurrence



A Recursion Tree for the Recurrence



A Recursion Tree for the Recurrence



A Recursion Tree for the Recurrence in the above recursion tree, each level has cost *cn*. The top level has cost *cn*. The next level down has 2 sub problems; each contributing cost *cn*/2.The next level has 4 sub problems, each contributing cost *cn*/4. Each time we go down one level, the number of sub problems doubles but the cost per sub problem halves. Therefore, cost per level stays the same. The height of this recursion tree is log *n* and there are log *n* + 1 levels. Total Running Time of a tree for a problem size of 2^i has log 2^i + 1 = *i* +1 levels. The fully expanded tree recursion tree has log n+1 levels. When n=1 than 1 level log 1=0, so correct number of levels log n+1. Because we assume that the problem size is a power of 2, the 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^i tree implying *i* + 2 levels. Since log 2^i + 1 = *i* + 2, we are done with the inductive argument. Total cost is sum of costs at each level of the tree. Since we have log *n* +1 levels, each costing *cn*, the total cost is *cn* log n + *cn*. Ignore low-order term of *cn* and constant coefficient *c*, and we have, Θ (*n* log *n*). The fully expanded tree has lg n +1 levels and each level contributes a total cost of cn. Therefore T (n)= cn log n + cn = θ(nlog n). Growth of Functions We look at input sizes large enough to make only the order of growth of the running time relevant.

2.3.4 Divide and Conquer: Quick Sort

Pick one element in the array, which will be the *pivot*. Make one pass through the array, called a *partition* step, re-arranging the entries so that, entries smaller than the pivot are to the left of the pivot. Entries larger than the pivot are to the right. Recursively apply quick sort to the part of the array that is to the left of the pivot, and to the part on its right. No merge step, at the end all the elements are in the proper order. Choosing the Pivot some fixed element: e.g. the first, the last, the one in the middle. Bad choice - may turn to be the smallest or the largest element, and then one of the partitions will be empty. Randomly chosen (by random generator) still a bad choice. The median of the array (if the array has N numbers, the median is the [N/2] largest number). This is difficult to compute - increases the complexity. The median-of-three choice: take the first, the last and the middle element. Choose the median of these three elements.

Quick Sort:
Quick sort is introduced by Hoare in the year 1962.
All elements to the left of pivot are smaller or equal than pivot, and
All elements to the right of pivot are greater or equal than pivot
Pivot in correct place in sorted array/list
Divide: Partition into sub arrays (sub-lists)
Conquer: Recursively sort 2 sub arrays
Combine: Trivial
Problem: Sort *n* keys in non-decreasing order
Inputs: Positive integer *n*, array of keys S indexed from 1 to n
Output: The array S containing the keys in non-decreasing order.

Quick sort (low, high)
1. if high > low
2. then partition(low, high, pivotIndex)
3. quick sort(low, pivotIndex -1)
4. quick sort(pivotIndex +1, high)

Partition array for Quick sort
partition (low, high, pivot)

1. pivotitem = S [low]
2. k=low
3. for j = low +1 to high
4. do if S [j] < pivotitem
5. then k = k + 1
6. exchange S [j] and S [k]
7. pivot = k
8. exchange S[low] and S[pivot]

Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
Divide: pick a random element x (called pivot) and partition S into
L elements less than x
E elements equal x
G elements greater than x
Recur: sort L and G
Conquer: join L, E and G

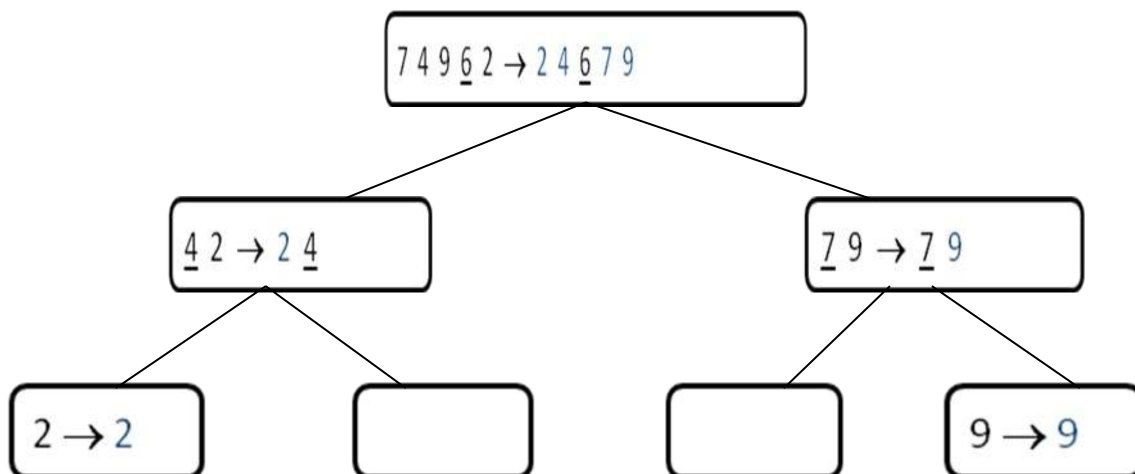
Partition

We partition an input sequence as follows:
-We remove, in turn, each element y from S and
-We insert y into L, E or G, depending on the result of the comparison with the pivot x
Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time. Thus, the partition step of quick-sort takes $O(n)$ time

Quick-Sort Tree

An execution of quick-sort is depicted by a binary tree. Each node represents a recursive call of quick-sort and stores. Unsorted sequence before the execution and its pivot. Sorted sequence at the end of the execution

- The root is the initial call
- The leaves are calls on subsequences of size 0 or 1



Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

The sub-arrays are stored in the original data array. Partitioning loops through, swapping elements below/above pivot.

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	100

Partitioning result

7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

<=data [pivot] ←　　　　　　　　→ >data [pivot]

Quick sort: Worst Case

Assume first element is chosen as pivot. Assume we get array that is already in order:

Pivot_index=0

2	4	10	12	13	50	57	63	100

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

Complexity of Quick Sort

If we have an array of equal elements, the array index will never increment i or decrement j, and will do infinite swaps. i and j will never cross.

Worst Case: O (N^2)

This happens when the pivot is the smallest (or the largest) element. Then one of the partitions is empty, and we repeat recursively the procedure for N-1 elements.

Worst-case Running Time

The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element. One of *L* and *G* has size *n* - 1 and the other has size 0

The running time is proportional to the sum

$n + (n - 1) + \ldots + 2 + 1$

Thus, the worst-case running time of quick-sort is $O(n^2)$

Depth	time
0	*N*
1	*n* - 1
...	...
n – 1	1

Worst-Case Analysis

The pivot is the smallest (or the largest) element

T (N) = T (N-1) + cN, N > 1

Telescoping:

T (N-1) = T (N-2) + c (N-1)

T (N-2) = T (N-3) + c (N-2)

T (N-3) = T (N-4) + c (N-3)

…………...

T (2) = T (1) + c.2

T (N) + T (N-1) + T (N-2) + … + T (2) =

= T (N-1) + T (N-2) + … + T (2) + T (1) +

C (N) + c (N-1) + c (N-2) + … + c.2

T (N) = T (1) +

c times (the sum of 2 thru N)

= T (1) + c (N (N+1) / 2 -1) = O (N^2)

Average-case: O (N logN)

Best-case: O (N logN)

The pivot is the median of the array, the left and the right parts have same size. There are logN partitions, and to obtain each partition we do N comparisons (and not more than N/2 swaps). Hence the complexity is O (NlogN).

Best case Analysis:

T (N) = T (i) + T (N - i -1) + cN

The time to sort the file is equal to the time to sort the left partition with i elements, plus the time to sort the right partition with N-i-1 elements, plus the time to build the partitions.

The pivot is in the middle

T (N) = 2 T (N/2) + cN

Divide by N: T (N) / N = T (N/2) / (N/2) + c

Telescoping:

T (N) / N = T (N/2) / (N/2) + c

T (N/2) / (N/2) = T (N/4) / (N/4) + c

T (N/4) / (N/4) = T (N/8) / (N/8) + c

……

T (2) / 2 = T (1) / (1) + c

Add all equations:

T (N) / N + T (N/2) / (N/2) + T (N/4) / (N/4) + …. + T (2) / 2 =

= (N/2) / (N/2) + T (N/4) / (N/4) + … + T (1) / (1) + c.logN

After crossing the equal terms:

T (N)/N = T (1) + c * LogN

T (N) = N + N * c * LogN = O (NlogN)

Advantages and Disadvantages:

Advantages

One of the fastest algorithms on average

Does not need additional memory (the sorting takes place in the array - this is called in-place processing)

Disadvantages

The worst-case complexity is O (N^2)

Applications

Commercial applications

Quick Sort generally runs fast

No additional memory

The above advantages compensate for the rare occasions when it runs with O (N^2)

2.3.5 Divide and Conquer: Selection Sort

Definition: First find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.

Selection sort is:

-The simplest sorting techniques.

-a good algorithm to sort a small number of elements

-an incremental algorithm – induction method

Selection sort is Inefficient for large lists.

Incremental algorithms → process the input elements one-by-one and maintain the solution for the elements processed so far.

Let A [1…n] be an array of n elements. A simple and straightforward algorithm to sort the entries in A works as follows. First, we find the minimum element and store it in A [1]. Next, we find the minimum of the remaining n-1 elements and store it in A [2]. We continue this way until the second largest element is stored in A [n-1].

Input: A [1…n];

Output: A [1…n] sorted in non-decreasing order;

1. for $i \leftarrow 1$ to n-1
2. $k \leftarrow i$;
3. for $j \leftarrow i+1$ to n
4. if $A[j]<A[k]$ then $k \leftarrow j$;
5. end for;
6. if $k \neq i$ then interchange $A[i]$ and $A[k]$;
7. end for;

Procedure of selection sort

i.Take multiple passes over the array.

ii.Keep already sorted array at high-end.

iii.Find the biggest element in unsorted part.

iv.Swap it into the highest position in unsorted part.

v.Invariant: each pass guarantees that one more element is in the correct position (same as bubble sort) a lot fewer swaps than bubble sort!

12	8	3	21	99	1	Start- unsorted

Pass 1

12	8	3	21	99	1
12	8	3	21	1	99

Pass 2

12	8	3	21	1	99
12	8	3	1	21	99

Pass 3

12	8	3	1	21	99

1	8	3	12	21	99

Pass 4

1	8	3	12	21	99
1	3	8	12	21	99

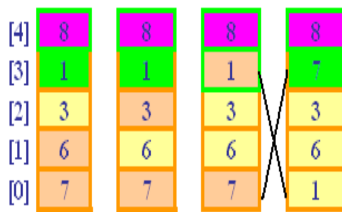
Pass 5

1	3	8	12	21	99

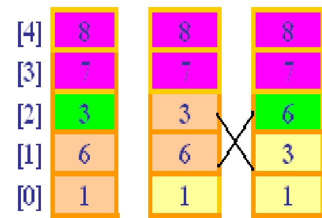
1	3	8	12	21	99

Sorted

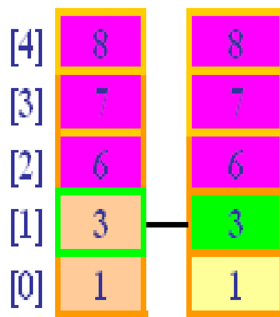
Example Execution of selection sort Tracing

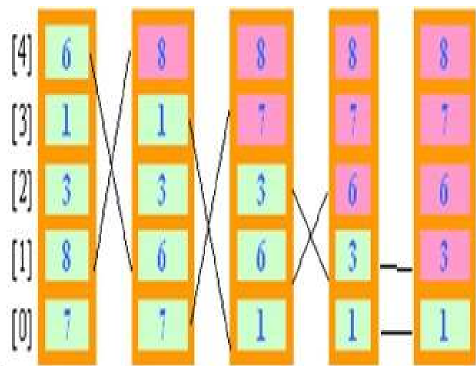


Pass 2
Last 3
Largest index 0, 0, 0
P=1, 2, 3



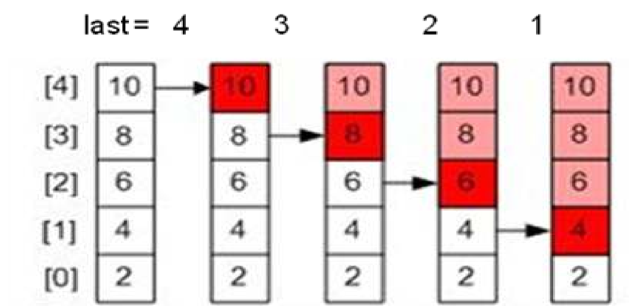
Pass 3
Last 2
largest index 0, 1
p=1, 2





Pass 4
last = 1
Largest Index = 0, 1
p = 1

Selection Sort Implementation for Best Case [2 4 6 8 10]



largestIndex = 4 3 2 1

Selection Sort Analysis

For an array with size n, the external loop will iterate from *n-1 to 1.*
for (int last = n-1; last>=1; --last) For each iteration, to find the largest number in sub array, the number of comparison inside the internal loop must is equal to the value of last. for (int p=1;p <=last; ++p) Therefore the total comparison for Selection Sort in each iteration is *(n-1) + (n-2) + ... 2 + 1*. Generally, the number of comparisons between elements in Selection Sort can be stated as follows:

$$(n-1)+(n-2)+.......+2+1=\frac{n(n-1)}{2}=O(n^2)$$

Selection Sort – Algorithm Complexity

Time Complexity for Selection Sort is the same for all cases - worst case, best case or average case O (n^2). The number of comparisons between elements is the same. The efficiency of Selection Sort does not depend on the initial arrangement of the data.

Alg.: SELECTION-SORT *(A)*	*Cost*	*times*
1. n ← length[A]	c1	1
2. for j ← 1 to n – 1	c2	n
3. do smallest ← j	c3	n-1
4. for i ← j + 1 to n		$\sum_{j=1}^{n-1}(n-j+1)$
5. do if A[i] < A[smallest]		$\sum_{j=1}^{n-1}(n-j)$
6.then smallest ← i		$\sum_{j=1}^{n-1}(n-j)$
7. exchange A[j] ↔ A[smallest]	c7	

2.11 Stressen's Matrix Multiplication:

Multiplication of Large Integer

Consider the problem of multiplying two (large) *n*-digit integers represented b arrays of their digits such as:

A = 12345678901357986429 B = 87654321284820912836

The grade-school algorithm:

a_1	a_2	…	a_n
b_1	b2	…	bn
(d_{10})	$d_{11}d_{12}$	…	d_{1n}
(d_{20})	$d_{21}d_{22}$	…	d_{2n}
… … … … … … …			
(d_{n0})	$d_{n1}d_{n2}$	…	d_{nn}

Efficiency: $\Theta(n^2)$ single-digit multiplications

First Divide-and-Conquer Algorithm

A small example: $A * B$ where $A = 2135$ and $B = 4014$

$A = (21 \cdot 10^2 + 35)$, $B = (40 \cdot 10^2 + 14)$

So, $A * B = (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$

$= 21 * 40 \cdot 10^{4} + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14$

In general, if $A = A_1A_2$ and $B = B_1B_2$ (where A and B are n-digit, A_1, A_2, B_1, B_2 are $n/2$-digit numbers),

$A * B = A_1 * B_1 \cdot 10^{n} + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

Recurrence for the number of one-digit multiplications M(*n*):

M (n) = 4 M (n/2),

M (1) = 1

Solution: $M(n) = n^2$

Second Divide-and-Conquer Algorithm:

$A * B = A_1 * B_1 \cdot 10^{n} + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

The idea is to decrease the number of multiplications from 4 to 3:

$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,$

I.e., $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2,$ which requires only 3 multiplications at the expense of (4-1) extra add/sub.

Recurrence for the number of multiplications M (n):

$M(n) = 3M(n/2)$, $M(1) = 1$

Solution: $M(n) = 3^{\log 2n} = n^{\log 23} \approx n^{1.585}$

Example of Large-Integer Multiplication:

2135 * 4014

(21*10^2 + 35) * (40*10^2 + 14)

= (21*40)*10^4 + c1*10^2 + 35*14
where c1 = (21+35)*(40+14) - 21*40 - 35*14, and
21*40 = (2*10 + 1) * (4*10 + 0)
= (2*4)*10^2 + c2*10 + 1*0
where c2 = (2+1)*(4+0) - 2*4 - 1*0, etc.
This process requires 9 digit multiplications as opposed to 16.

Conventional Matrix Multiplication:
Brute-force algorithm

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

$$= \begin{pmatrix} a_{00} * b_{00} + a_{01} * b_{10} & a_{00} * b_{01} + a_{01} * b_{11} \\ a_{10} * b_{00} + a_{11} * b_{10} & a_{10} * b_{01} + a_{11} * b_{11} \end{pmatrix}$$

8 multiplications, 4 additions   Efficiency class in general: $\Theta(n^3)$

Strassen's Matrix Multiplication
Strassen's algorithm for two 2x2 matrices (1969):

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

$$= \begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{pmatrix}$$

$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$
$m_2 = (a_{10} + a_{11}) * b_{00}$
$m_3 = a_{00} * (b_{01} - b_{11})$
$m_4 = a_{11} * (b_{10} - b_{00})$
$m_5 = (a_{00} + a_{01}) * b_{11}$
$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$
$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

7 multiplications, 18 additions

Strassen observed [1969] that the product of two matrices can be computed in general as follows

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$= \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}$$

Formulas for Strassen's Algorithm

$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$

$M_2 = (A_{10} + A_{11}) * B_{00}$

$M_3 = A_{00} * (B_{01} - B_{11})$

$M_4 = A_{11} * (B_{10} - B_{00})$

$M_5 = (A_{00} + A_{01}) * B_{11}$

$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$

$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$

Analysis of Strassen's Algorithm

If n is not a power of 2, matrices can be padded with zeros.

<u>Number of multiplications</u>:

$$M(n) = 7\,M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 7^{\log 2n} = n^{\log 27} \approx n^{2.807}$ vs. n^3 of brute-force alg.

Algorithms with better asymptotic efficiency are known but they are even more complex and not used in practice.

******