CONTENTS

MODULE - I

- Lecture 1 - Introduction to Design and analysis of algorithms
- Lecture 2 - Growth of Functions (Asymptotic notations)
- Lecture 3 - Recurrences, Solution of Recurrences by substitution
- Lecture 4 - Recursion tree method
- Lecture 5 - Master Method
- Lecture 6 - Design and analysis of Divide and Conquer Algorithms
- Lecture 7 - Worst case analysis of merge sort, quick sort and binary search
- Lecture 8 - Heaps and Heap sort
- Lecture 9 - Priority Queue
- Lecture 10 - Lower Bounds for Sorting

Lecture 1 - Introduction to Design and analysis of algorithms

What is an algorithm?

Algorithm is a *set of steps to complete a task.*

For example,

Task: to make a cup of tea.

Algorithm:

- add water and milk to the kettle,
- boilit, add tea leaves,
- Add sugar, and then serve it in cup.

What is *Computer algorithm*?

''a set of steps to accomplish or complete a task that is described precisely enough that a computer can run it''.

Described precisely: very difficult for a machine to know how much water, milk to be added etc. in the above tea making algorithm.

These algorithmsrun on computers or computational devices.Forexample, GPS in our smartphones, Google hangouts.

GPS uses *shortest path algorithm.* Online shopping uses cryptography which uses RSA algorithm.

Characteristics of an algorithm:-

- Must take an input.
- Must give some output(yes/no,valueetc.)
- Definiteness –each instruction is clear and unambiguous.
- Finiteness –algorithm terminates after a finite number of steps.
- Effectiveness –every instruction must be basic i.e. simple instruction.

Expectation from an algorithm

- Correctness:-
  - Correct: Algorithms must produce correct result.
  - Produce an incorrect answer:Even if it fails to give correct results all the time still there is a control on how often it gives wrong result. Eg.Rabin-Miller PrimalityTest (Used in RSA algorithm): It doesn't give correct answer all the time.1 out of 2^{50} times it gives incorrect result.
  - Approximation algorithm: Exact solution is not found, but near optimal solution can be found out. (Applied to optimization problem.)
- Less resource usage:

  Algorithms should use less resources (time and space).

Resource usage:

Here, the time is considered to be the primary measure of efficiency .We are also concerned with how much the respective algorithm involves the computer memory.But mostly time is the resource that is dealt with. And the actual running time depends on a variety of backgrounds: like the speed of the Computer, the language in which the algorithm is implemented, the compiler/interpreter, skill of the programmers etc.

So, mainly the resource usage can be divided into: 1.Memory (space) 2.Time

Time taken by an algorithm?

- performance measurement or Apostoriori Analysis: Implementing the algorithm in a machine and then calculating the time taken by the system to execute the program successfully.
- Performance Evaluation or Apriori Analysis. Before implementing the algorithm in a system. This is done as follows

1. How long the algorithm takes :-will be represented as a function of the size of the input.

f(n)→how long it takes if 'n' is the size of input.

2. How fast the function that characterizes the running time grows with the input size.
   "Rate of growth of running time".
   The algorithm with less rate of growth of running time is considered better.

How algorithm is a technology ?

Algorithms are just like a technology. We all use latest and greatest processors but we need to run implementations of good algorithms on that computer in order to properly take benefits of our money that we spent to have the latest processor. Let's make this example more concrete by pitting a faster computer(computer A) running a sorting algorithm whose running time on n values grows like n^2 against a slower computer (computer B) running asorting algorithm whose running time grows like n lg n. They eachmust sort an array of 10 million numbers. Suppose that computer A executes 10 billion instructions per second (faster than anysingle sequential computer at the time of this writing) and computer B executes only 10 million instructions per second, so that computer A is1000 times faster than computer B in raw computing power. To makethe difference even more dramatic, suppose that the world's craftiestprogrammer codes in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers. Suppose furtherthat just an average programmer writes for computer B, using a high-level language with an inefficient compiler, with the resulting code taking 50n lg n instructions.

Computer A (Faster)	Computer B(Slower)
Running time grows like n^2.	Grows innlogn.
10 billion instructions per sec.	10million instruction per sec
$2n^2$ instruction.	50 nlogn instruction.

Time taken=$\frac{2\times(10^7)^2}{10^{10}} = 20{,}000$ $\frac{50\times10^7\times\log 10^7}{10^7} \approx 1163$

It is more than 5.5hrs it is under 20 mins.

So choosing a good algorithm (algorithm with slower rate of growth) as used by computer B affects a lot.

Lecture 2 - Growth of Functions (Asymptotic notations)

Before going for growth of functions and asymptotic notation let us see how to analyase an algorithm.

How to analyse an Algorithm

Let us form an algorithm for Insertion sort (which sort a sequence of numbers).The pseudo code for the algorithm is give below.

Pseudo code:

for j=2 to A length ---------- C_1

key=A[j]----------C_2

//Insert A[j] into sorted Array A[1.....j-1]----------C_3

i=j-1----------C_4

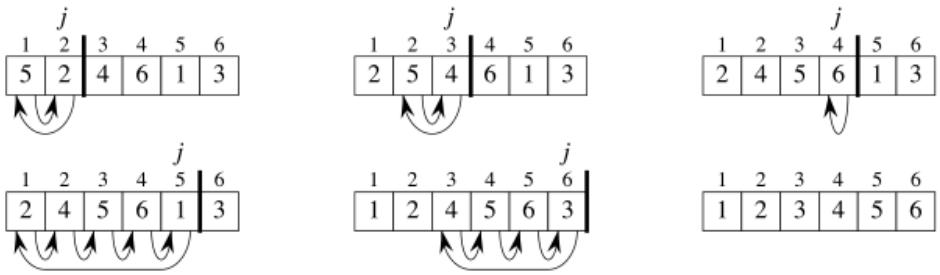
while i>0 & A[j]>key----------C_5

A[i+1]=A[i]----------C_6

i=i-1----------C_7

A[i+1]=key----------C_8

Example:



Let C_i be the cost of i^{th} line. Since comment lines will not incur any cost $C_3=0$.

Cost	No. Of times Executed
C_1	n
C_2	n-1
$C_3=0$	n-1
C_4	n-1
C_5	$\sum_{j=2}^{n-1} t_j$
C_6	$\sum_{j=2}^{n}(t_j - 1)$
C_7	$\sum_{j=2}^{n}(t_j - 1)$
C_8	n-1

Running time of the algorithm is:

$T(n)=C_1n+C_2(n-1)+0(n-1)+C_4(n-1)+C_5(\sum_{j=2}^{n-1} t_j)+C_6(\sum_{j=2}^{n} t_j - 1)+C_7(\sum_{j=2}^{n} t_j - 1)+ C_8(n-1)$

Best case:

It occurs when Array is sorted.

All $t_j$values are 1.

$T(n)=C_1n+C_2(n-1)+0\ (n-1)+C_4(n-1)+C_5(\sum_{j=2}^{n-1} 1)+C_6(\sum_{j=2}^{n} 0)+C_7(\sum_{j=2}^{n} 0)+ C_8(n-1)$

$=C_1n+C_2\ (n-1)+0\ (n-1)+C_4\ (n-1)+C_5(n-1)+ C_8\ (n-1)$

$= (C_1+C_2+C_4+C_5+ C_8)\ n-(C_2+C_4+C_5+ C_8)$

- Which is of the forman+b.
- Linear function of n.So, linear growth.

Worst case:

It occurs when Array is reverse sorted, and t_j =j

$T(n)=C_1n+C_2(n-1)+0\ (n-1)+C_4(n-1)+C_5(\sum_{j=2}^{n-1} j)+C_6(\sum_{j=2}^{n} j-1)+C_7(\sum_{j=2}^{n} j-1)+ C_8(n-1)$

$=C_1n+C_2(n-1)+C_4(n-1)+C_5(\frac{n(n-1)}{2}-1)+C_6(\sum_{j=2}^{n} \frac{n(n-1)}{2})+C_7(\sum_{j=2}^{n} \frac{n(n-1)}{2})+ C_8(n-1)$

which is of the form an^2+bn+c

Quadratic function. So in worst case insertion set grows in n^2.

Why we concentrate on worst-case running time?

- The worst-case running time gives a guaranteed upper bound on the runningtime for any input.
- For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.
- Why not analyze the average case? Because it's often about as bad as the worst case.

Order of growth:

It is described by the highest degree term of the formula for running time. (Drop lower-order terms. Ignore the constant coefficient in the leading term.)

Example: We found out that for insertion sort the worst-case running time is of the form $an^2 + bn + c$.

Drop lower-order terms. What remains is an^2.Ignore constant coefficient. It results in n^2.But we cannot say that the worst-case running time T(n) equals n^2 .Rather It grows like n^2 . But it doesn't equal n^2.We say that the running time is $\Theta(n^2)$ to capture the notion that the order of growthis n^2.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.

Asymptotic notation

- It is a way to describe the characteristics of a function in the limit.
- It describes the rate of growth of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- It is a way to compare "sizes" of functions:

  O≈ ≤

  Ω≈ ≥

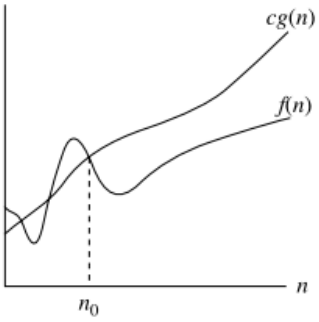
  Θ ≈ =

  o ≈ <

  ω ≈ >

O-notation

$O(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that
$0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$.



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

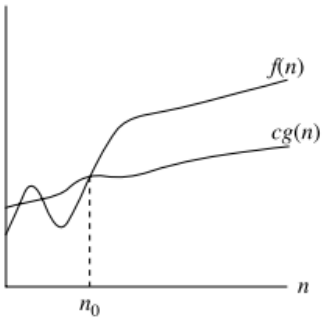
Example: $2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

Examples of functions in $O(n^2)$:

n^2
$n^2 + n$
$n^2 + 1000n$
$1000n^2 + 1000n$
Also,
n
$n/1000$
$n^{1.99999}$
$n^2/\lg\lg\lg n$

Ω-notation

$\Omega(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$.



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

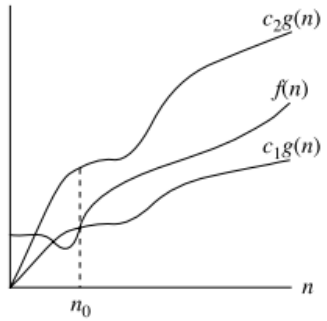
Example: $\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

Examples of functions in $\Omega(n^2)$:

n^2
$n^2 + n$
$n^2 - n$
$1000n^2 + 1000n$
$1000n^2 - 1000n$
Also,
n^3
$n^{2.00001}$
$n^2 \lg\lg\lg n$
2^{2^n}

Θ-notation

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$



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

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

***o*-notation**

$$o(g(n)) = \{f(n) : \text{for all constants } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0\} .$$

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

$n^{1.9999} = o(n^2)$
$n^2/\lg n = o(n^2)$
$n^2 \neq o(n^2)$ (just like $2 \not< 2$)
$n^2/1000 \neq o(n^2)$

***ω*-notation**

$$\omega(g(n)) = \{f(n) : \text{for all constants } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0\} .$$

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

$n^{2.0001} = \omega(n^2)$
$n^2 \lg n = \omega(n^2)$
$n^2 \neq \omega(n^2)$

Lecture 3-5: Recurrences, Solution of Recurrences by substitution, Recursion Tree and Master Method

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

• If the given instance of the problem is small or simple enough, just solve it.

• Otherwise, reduce the problem to one or more simpler instances of the same problem.

Recursion is generally expressed in terms of recurrences. In other words, when an algorithm calls to itself, we can often describe its running time by a **recurrence equation** which describes the overall running time of a problem of size n in terms of the running time on smaller inputs.

E.g.the worst case running time T(n) of the merge sort procedure by recurrence can be expressed as

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

whose solution can be found as $T(n)=\Theta(n \log n)$

There are various techniques to solve recurrences.

1. **SUBSTITUTION METHOD**:

The substitution method comprises of 3 steps

i. Guess the form of the solution
ii. Verify by induction
iii. Solve for constants

We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values. Hence the name "substitution method". This method is powerful, but we must be able to guess the form of the answer in order to apply it.

e.g.recurrence equation: $T(n)=4T(n/2)+n$

step 1: guess the form of solution

T(n)=4T(n/2)

⇨ F(n)=4f(n/2)

⇨ F(2n)=4f(n)

⇨ $F(n)=n^2$

So, T(n) is order of n^2

Guess $T(n)=O(n^3)$

Step 2: verify the induction

Assume $T(k)<=ck^3$

T(n)=4T(n/2)+n

$<=4c(n/2)^3+n$

$<=cn^3/2+n$

$<=cn^3-(cn^3/2-n)$

$T(n)<=cn^3$ as $(cn^3/2 –n)$ is always positive

So what we assumed was true.

⇨ $T(n)=O(n^3)$

Step 3: solve for constants

$Cn^3/2-n>=0$

⇨ n>=1

⇨ c>=2

Now suppose we guess that $T(n)=O(n^2)$ which is tight upper bound

Assume,$T(k)<=ck^2$

so,we should prove that $T(n)<=cn^2$

T(n)=4T(n/2)+n

⇨ $4c(n/2)^2+n$

⇨ cn^2+n

So,T(n) will never be less than cn^2. But if we will take the assumption of $T(k)=c_1 k^2-c_2k$, then we can find that $T(n) = O(n^2)$

2. BY ITERATIVE METHOD:

e.g. $T(n)=2T(n/2)+n$

$=> 2[2T(n/4) + n/2]+n$

$=>2^2T(n/4)+n+n$

$=> 2^2[2T(n/8)+ n/4]+2n$

$=>2^3T(n/2^{3)} +3n$

After k iterations ,$T(n)=2^kT(n/2^k)+kn$----------(1)

Sub problem size is 1 after $n/2^k=1 => k=logn$

So,afterlogn iterations ,the sub-problem size will be 1.

So,when k=logn is put in equation 1

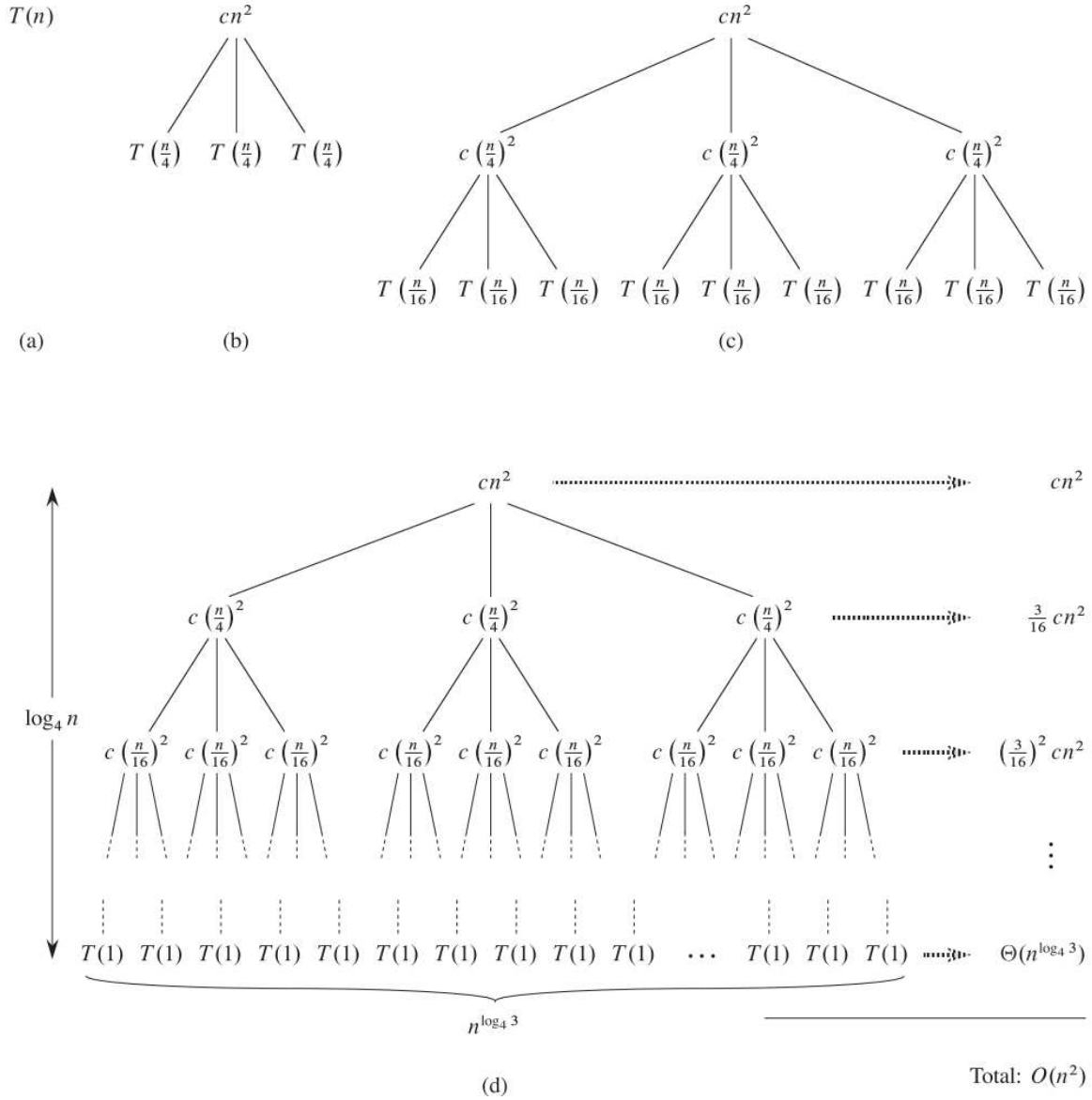
$T(n)=nT(1)+nlogn$

- ⇨ $nc+nlogn$ (say $c=T(1)$)
- ⇨ $O(nlogn)$

3.BY RECURSSION TREE METHOD:

In a recursion tree ,each node represents the cost of a single sub-problem somewhere in the set of recursive problems invocations .we sum the cost within each level of the tree to obtain a set of per level cost,and then we sum all the per level cost to determine the total cost of all levels of recursion .

Constructing a recursion tree for the recurrence $T(n)=3T(n/4)+cn^2$



Constructing a recursion tree for the recurrence T (n)= 3T (n=4) + cn^2.. Part (a) shows T (n), which progressively expands in (b)–(d) to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

Sub problem size at depth i =$n/4^i$

Sub problem size is 1 when $n/4^i=1$ => $i=\log_4 n$

So, no. of levels =$1+ \log_4 n$

Cost of each level = (no. of nodes) x (cost of each node)

No. Of nodes at depth i=3^i

Cost of each node at depth i=c $(n/4^i)^2$

Cost of each level at depth i=3^i c $(n/4^i)^2$ = $(3/16)^i cn^2$

T(n)= $\sum_{i=0}^{\log_4 n} cn^2(3/16)^i$

T(n)= $\sum_{i=0}^{\log_4 n -1} cn^2(3/16)^i$ + cost of last level

Cost of nodes in last level =3^iT(1)

- ⇨ $c3^{\log_4 n}$ (at last level i=$\log_4 n$)
- ⇨ $cn^{\log_4 3}$

T(n)= $\sum_{i=0}^{\log_4 n -1} cn^2(3/16)^i$ + $c\, n^{\log_4 3}$

<= $cn^2 \sum_{i=0}^{\infty} (3/16)^i + cn^{\log_4 3}$

⇨ <= cn^2*(16/13)+ $cn^{\log_4 3}$ => T(n)=O(n^2)

4.BY MASTER METHOD:

The master method solves recurrences of the form

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

where a>=1 and b>1 are constants and f(n) is a asymptotically positive function .

To use the master method, we have to remember 3 cases:

1. If f(n)=O($n^{\log_b a - \varepsilon}$) for some constants Ɛ >0,then T(n)=Θ($n^{\log_b a}$)

2. If f(n)=Θ($n^{\log_b a}$) then T(n)=Θ($n^{\log_b a}$logn)

3. If f(n)=Ω($n^{\log_b a+\varepsilon}$) for some constant Ɛ>0 ,and if a*f(n/b)<=c*f(n) for some constant c<1 and all sufficiently large n,then T(n)=Θ(f(n))

e.g. $T(n)=2T(n/2)+n\log n$

ans: a=2 b=2

f(n)=nlogn

using 2nd formula

$f(n)=\Theta(n^{\log_2 2}\log^k n)$

$=>\Theta(n^1 \log^k n)=n\log n$ =>K=1

$T(n)=\Theta(n^{\log_2 2} \log^1 n)$

$=>\Theta(n\log^2 n)$

Lecture 6 - Design and analysis of Divide and Conquer Algorithms

DIVIDE AND CONQUER ALGORITHM

- In this approach ,we solve a problem recursively by applying 3 steps
  1. **DIVIDE**-break the problem into several sub problems of smaller size.
  2. **CONQUER**-solve the problem recursively.
  3. **COMBINE**-combine these solutions to create a solution to the original problem.

CONTROL ABSTRACTION FOR DIVIDE AND CONQUER ALGORITHM

```
Algorithm D and C (P)
{
    if small(P)
        then return S(P)
    else
        { divide P into smaller instances P1 ,P2 .....Pk
          Apply D and C to each sub problem
            Return combine (D and C(P1)+ D and C(P2)+.....+D and C(Pk))
    }
```

}

Let a recurrence relation is expressed as T(n)=

Θ(1), if n<=C

aT(n/b) + D(n)+ C(n) ,otherwise

then n=input size a=no. Of sub-problemsn/b= input size of the sub-problems

Lecture 7: Worst case analysis of merge sort, quick sort

Merge sort

It is one of the well-known divide-and-conquer algorithm. This is a simple and very efficient algorithm for sorting a list of numbers.

We are given a sequence of n numberswhich we will assume is stored in an array A [1...n]. Theobjective is to output a permutation of this sequence, sorted in increasing order. This is normally done by permuting the elements within the array A.

How can we apply divide-and-conquer to sorting? Here are the major elements of the Merge Sort algorithm.

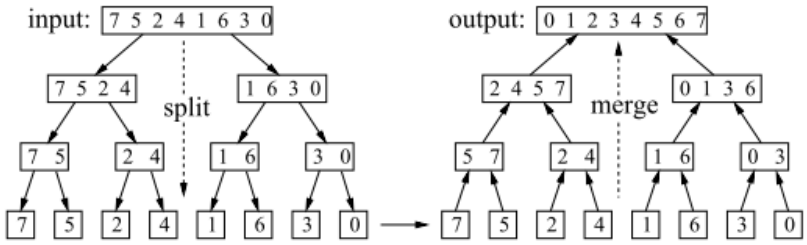
Divide: Split A down the middle into two sub-sequences, each of size roughly n/2 .

Conquer: Sort each subsequence (by calling MergeSort recursively on each).

Combine: Merge the two sorted sub-sequences into a single sorted list.

The dividing process ends when we have split the sub-sequences down to a single item. A sequence of length one is trivially sorted. The key operation where all the work is done is in the combine stage,which merges together two sorted lists into a single sorted list. It turns out that the merging process is quite easy to implement.

The following figure gives a high-level view of the algorithm. The “divide” phase is shown on the left. It works top-down splitting up the list into smaller sublists. The “conquer and combine” phases areshown on the right. They work bottom-up, merging sorted lists together into larger sorted lists.



Merge Sort

Designing the Merge Sort algorithm top-down. We’ll assume that the procedure thatmerges two sortedlist is available to us. We’ll implement it later. Because the algorithm is called recursively on sublists,in addition to passing in the array itself, we will pass in two indices, which indicate the first and lastindices of the subarray that we are to sort. The call MergeSort(A, p, r) will sort the sub-arrayA [p..r] and return the sorted result in the same subarray.

Here is the overview. If r = p, then this means that there is only one element to sort, and we may returnimmediately. Otherwise (if p < r) there are at least two elements, and we will invoke the divide-and-conquer. We find the index q, midway between p and r, namely q = (p + r) / 2 (rounded down to thenearest integer). Then we split the array into subarrays A [p..q] and A [q + 1 ..r] . Call Merge Sort recursively to sort each subarray. Finally, we invoke a procedure (which we have yet to write) whichmerges these two subarrays into a single sorted array.

```
MergeSort(array A, int p, int r) {
        if      (p < r) {                                   // we have at least 2 items
        q = (p + r)/2
        MergeSort(A, p, q)                                  // sort A[p..q]
        MergeSort(A, q+1, r)                                // sort A[q+1..r]
```

```
    Merge(A, p, q, r)                    // merge everything together
    }                                                                    }
```

Merging: All that is left is to describe the procedure that merges two sorted lists. Merge(A, p, q, r)assumes that the left subarray, A [p..q] , and the right subarray, A [q + 1 ..r] , have already been sorted.We merge these two subarrays by copying the elements to a temporary working array called B. Forconvenience, we will assume that the array B has the same index range A, that is, B [p..r] . We have to indices i and j, that point to the current elements ofeach subarray. We move the smaller element into the next position of B (indicated by index k) andthen increment the corresponding index (either i or j). When we run out of elements in one array, thenwe just copy the rest of the other array into B. Finally, we copy the entire contents of B back into A.

```
Merge(array A, int p, int q, int r) {          // merges A[p..q] with A[q+1..r]
    array B[p..r]
    i = k = p                                  //initialize     pointers
    j = q+1
    while (i <= q and j <= r) {                // while both subarrays are nonempty
    if (A[i] <= A[j]) B[k++] = A[i++]          // copy from left subarray
    else B[k++] = A[j++]                       // copy from right subarray
    }
    while (i <= q) B[k++] = A[i++]             // copy any leftover to B
    while (j <= r) B[k++] = A[j++]
    for i = p to r do A[i] = B[i]          // copy B back to A                    }
```

Analysis: What remains is to analyze the running time of MergeSort. First let us consider the running timeof the procedure Merge(A, p, q, r). Let n = r − p + 1 denote the total length of both the leftand right subarrays. What is the running time of Merge as a function of n? The algorithm contains fourloops (none nested in the other). It is easy to see that each loop can be executed at most n times. (Ifyou are a bit more careful you can actually see that all the while-loops

together can only be executed ntimes in total, because each execution copies one new element to the array B, and B only has space forn elements.) Thus the running time to Merge n items is $\Theta (n)$. Let us write this without the asymptoticnotation, simply as n. (We'll see later why we do this.)

Now, how do we describe the running time of the entire MergeSort algorithm? We will do this throughthe use of a recurrence, that is, a function that is defined recursively in terms of itself. To avoidcircularity, the recurrence for a given value of n is defined in terms of values that are strictly smallerthan n. Finally, a recurrence has some basis values (e.g. for $n = 1$), which are defined explicitly.

Let's see how to apply this to MergeSort. Let $T (n)$ denote the worst case running time of MergeSort onan array of length n. For concreteness we could count whatever we like: number of lines of pseudocode,number of comparisons, number of array accesses, since these will only differ by a constant factor.Since all of the real work is done in the Merge procedure, we will count the total time spent in theMerge procedure.

First observe that if we call MergeSort with a list containing a single element, then the running time is aconstant. Since we are ignoring constant factors, we can just write $T (n) =1$. When we call MergeSortwith a list of length $n >1$, e.g. Merge(A, p, r), where $r - p +1 = n$, the algorithm first computes $q = (p + r) / 2$. The subarray $A [p..q]$, which contains $q - p + 1$ elements. You can verify that is of size $n/ 2$. Thus the remaining subarray $A [q +1 ..r]$ has $n/ 2$ elements in it. How long does it taketo sort the left subarray? We do not know this, but because $n/ 2< n$ for $n >1$, we can express this as $T (n/ 2)$. Similarly, we can express the time that it takes to sort the right subarray as $T (n/ 2)$.

Finally, to merge both sorted lists takes n time, by the comments made above. In conclusion we have

$T (n) =1$ if $n = 1$,

$2T (n/ 2) + n$ otherwise.

Solving the above recurrence we can see that merge sort has a time complexity of $\Theta (n \log n)$.

QUICKSORT

- Worst-case running time: $O(n2)$.
- Expected running time: $O(n \lg n)$.
- Sorts in place.

Description of quicksort

Quicksort is based on the three-step process of divide-and-conquer.

• To sort the subarray$A[p \ldots r]$:

Divide: Partition $A[p \ldots r]$, into two (possibly empty) subarrays$A[p \ldots q-1]$ and $A[q+1 \ldots r]$, such that each element in the Þrstsubarray$A[p \ldots q-1]$ is $\leq A[q]$ and $A[q]$ is $\leq$ each element in the second subarray$A[q+1 \ldots r]$.

Conquer: Sort the two subarrays by recursive calls to QUICKSORT.

Combine: No work is needed to combine the subarrays, because they are sorted in place.

• Perform the divide step by a procedure PARTITION, which returns the index q that marks the position separating the subarrays.

```
QUICKSORT (A, p, r)
if p < r
then q ←PARTITION(A, p, r )
        QUICKSORT (A, p, q − 1)
        QUICKSORT (A, q + 1, r)

Initial call is QUICKSORT (A, 1, n)
```

Partitioning

Partition subarray$A[p \ldots r]$ by the following procedure:

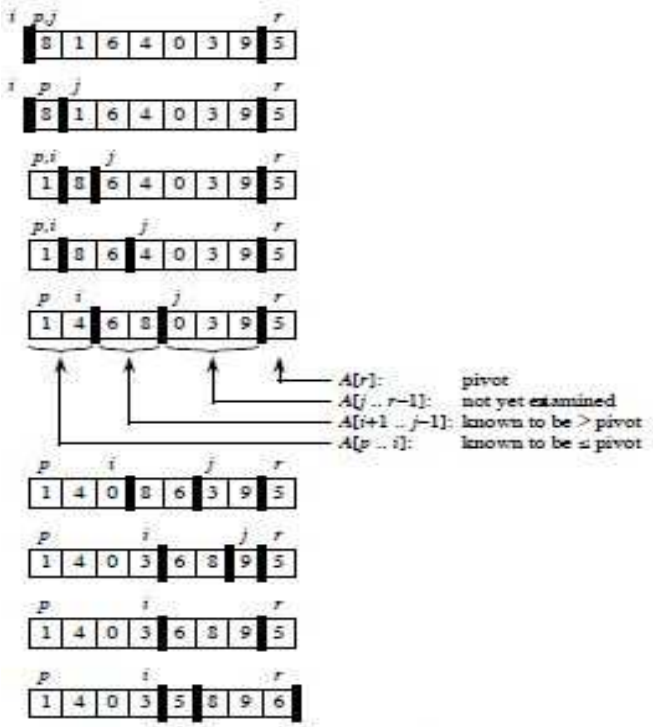
```
PARTITION (A, p, r)
  x ← A[r ]
  i ← p –1
  for j ← p to r –1
do if A[ j ] ≤ x

then i ← i + 1
```

exchange$A[i] \leftrightarrow A[j]$

exchange$A[i+1] \leftrightarrow A[r]$

return $i + 1$

- PARTITION always selects the last element $A[r]$ in the subarray $A[p \,..\, r]$ as the ***pivot*** the element around which to partition.
- As the procedure executes, the array is partitioned into four regions, some of which may be empty:



[The index j disappears because it is no longer needed once the for loop is exited.]

Performance of Quicksort

The running time of Quicksort depends on the partitioning of the subarrays:

• If the subarrays are balanced, then Quicksort can run as fast as mergesort.

• If they are unbalanced, then Quicksort can run as slowly as insertion sort.

Worst case

• Occurs when the subarrays are completely unbalanced.

• Have 0 elements in one subarray and $n - 1$ elements in the other subarray.

• Get the recurrence

$T(n) = T(n-1) + T(0) + \Theta(n)$

$= T(n-1) + \Theta(n)$

$= O(n2)$.

• Same running time as insertion sort.

• In fact, the worst-case running time occurs when Quicksort takes a sorted array as input, but insertion sort runs in $O(n)$ time in this case.

Best case

• Occurs when the subarrays are completely balanced every time.

• Each subarray has $\leq n/2$ elements.

• Get the recurrence

$T(n) = 2T(n/2) + \Theta(n) = O(n \lg n)$.

Balanced partitioning

• QuickPort's average running time is much closer to the best case than to the worst case.

• Imagine that PARTITION always produces a 9-to-1 split.

• Get the recurrence

$T(n) \leq T(9n/10) + T(n/10) + \_(n) = O(n \lg n)$.

• Intuition: look at the recursion tree.

• It's like the one for $T(n) = T(n/3) + T(2n/3) + O(n)$.

• Except that here the constants are different; we get log10 n full levels and log10/9 n levels that are nonempty.

• As long as it's a constant, the base of the log doesn't matter in asymptotic notation.

• Any split of constant proportionality will yield a recursion tree of depth $O(\lg n)$.

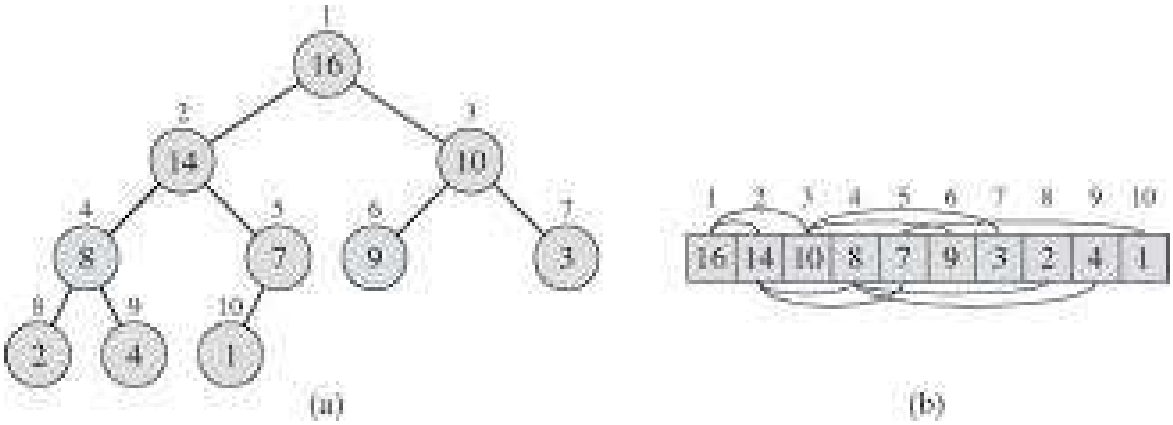
Lecture 8 - Heaps and Heap sort

HEAPSORT

- Inplace algorithm
- Running Time: O(n log n)
- Complete Binary Tree

The ***(binary) heap*** data structure is an array object that we can view as a nearly complete binary tree.Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

The root of the tree is A[1], and given the index *i* of a node, we can easily compute the indices of its parent, left child, and right child:



- PARENT (i) => ***return*** [i/2]
- LEFT (i) => ***return*** 2*i*
- RIGHT (i) => ***return*** 2i+ 1

On most computers, the LEFT procedure can compute 2i in one instruction by simply shifting the binary representation of i left by one bit position.

Similarly, the RIGHT procedure can quickly compute 2i + 1 by shifting the binary representation of i left by one bit position and then adding in a 1 as the low-order bit.

The PARENT procedure can compute *[i/2]* by shifting i right one bit position. Good implementations of heapsort often implement these procedures as "macros" or "inline" procedures.

There are two kinds of binary heaps: **max-heaps** and **min-heaps**.

- In a ***max-heap,***the ***max-heap property*** is that for every node *i* other than the root, **A[PARENT(i)] >= A[i] ,**that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.
- A ***min-heap*** is organized in the opposite way; the ***min-heap property*** is that for every node i other than the root, **A[PARENT(i)<=A[i] ,**

  The smallest element in a min-heap is at the root.

✓ The height of a node in a heap is the number of edges on the longest simple downward path from the node to a leaf and

✓ The height of the heap is the height of its root.

✓ Height of a heap of n elements which is based on a complete binary tree is **O(log n).**

Maintaining the heap property

MAX-HEAPIFY lets the value at A[i] "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

MAX-HEAPIFY(A,i)

1. l ← LEFT(i)
2. r ←RIGHT(i)
3. if A[l] > A[i]
4. largest ←l
5. if A[r] > A[largest]
6. Largest ←r
7. if largest != i
8. Then exchange A[i] ←→A[largest]

9. MAX-HEAPIFY(A,largest)

At each step, the largest of the elements A[i], A[LEFT(i)], and A[RIGHT(i)] is determined, and its index is stored in *largest.* If A[i] is largest, then the subtree rooted at node i is already a max-heap and the procedure terminates. Otherwise, one of the two children has the largest element, and A[i] is swapped with A[largest], which causes node i and its children to satisfy the max-heap property. The node indexed by *largest,* however, now has the original value A[i], and thus the subtree rooted at *largest* might violate the max-heap property. Consequently, we call MAX-HEAPIFY recursively on that subtree.

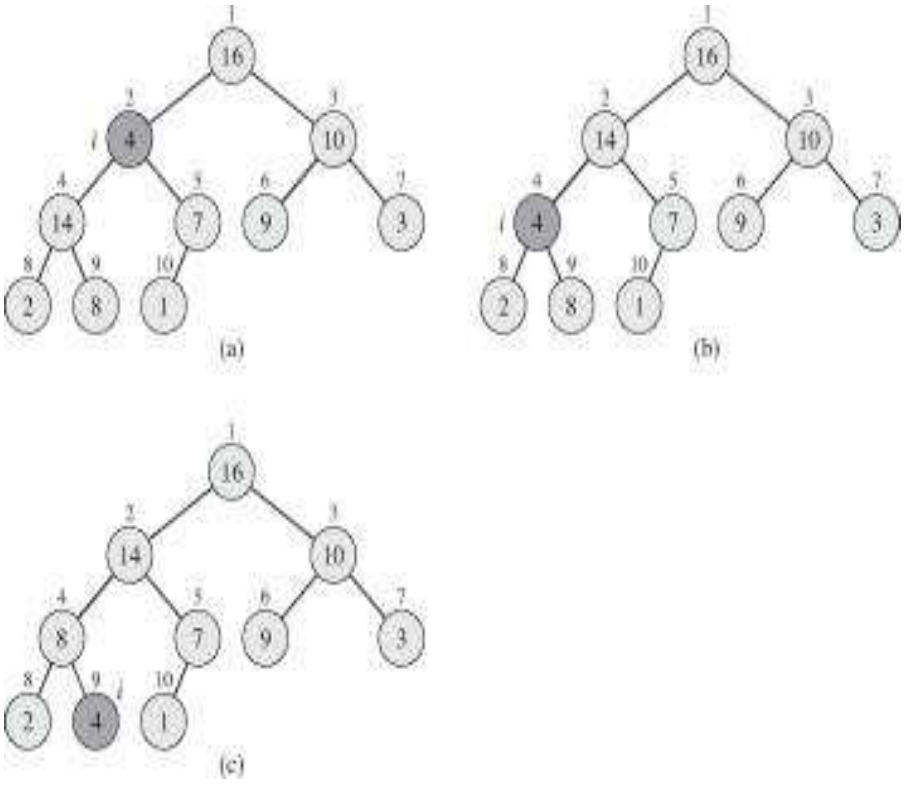


Figure: The action of MAX-HEAPIFY (A, 2), where *heap-size* = 10. **(a)** The initial configuration, with A [2] at node i = 2 violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging A [2] with A[4], which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY (A,4)

now has i = 4. After swapping A[4] with A[9], as shown in **(c),** node 4 is fixed up, and the recursive call MAX-HEAPIFY(A, 9) yields no further change to the data structure.

The running time of MAX-HEAPIFY by the recurrence can be described as

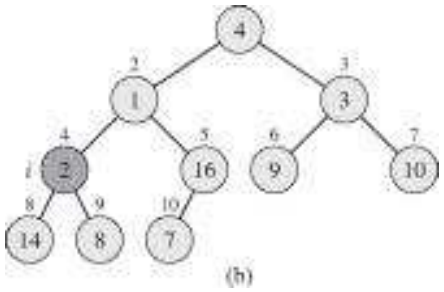
T (n) < = T (2n/3) + O (1)

The solution to this recurrence is T(n)=O(log n)

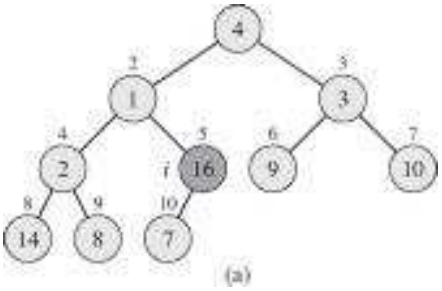
Building a heap

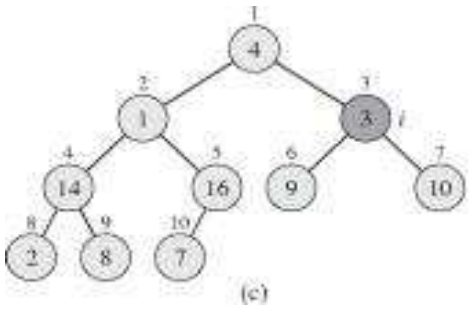
Build-Max-Heap(A)

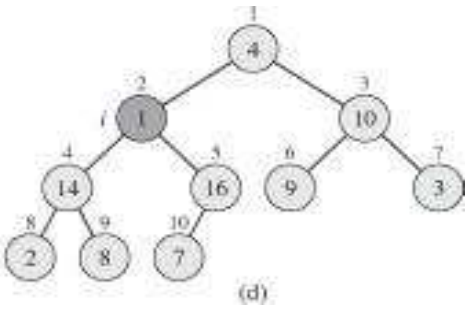
1. for i←[n/2] to 1
2. do MAX-HEAPIFY(A,i)

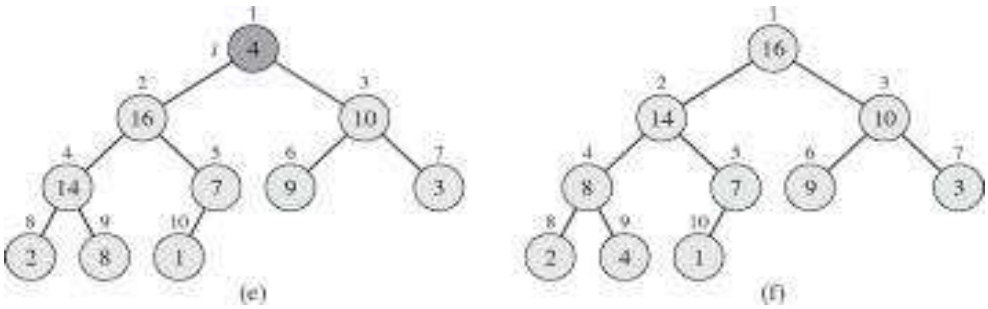


4	1	3	2	1	9	1	1	8	7









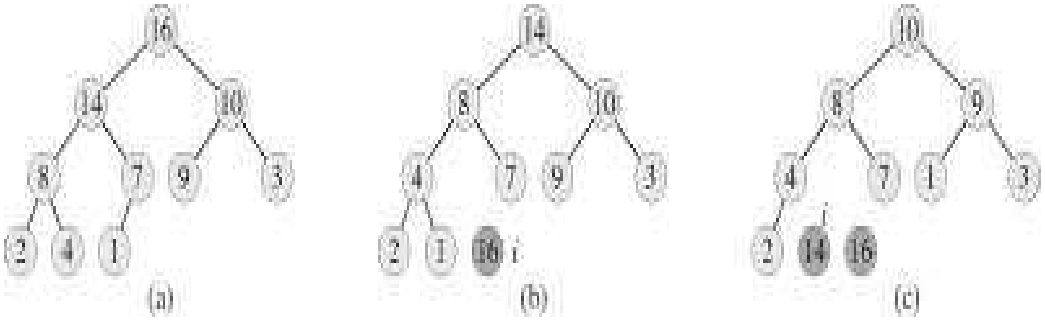
We can derive a tighter bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small. Our tighter analysis relies on the properties that an n-element heap has height [log n] and at most $[n/2^{h+1}]$ nodes of any height h.

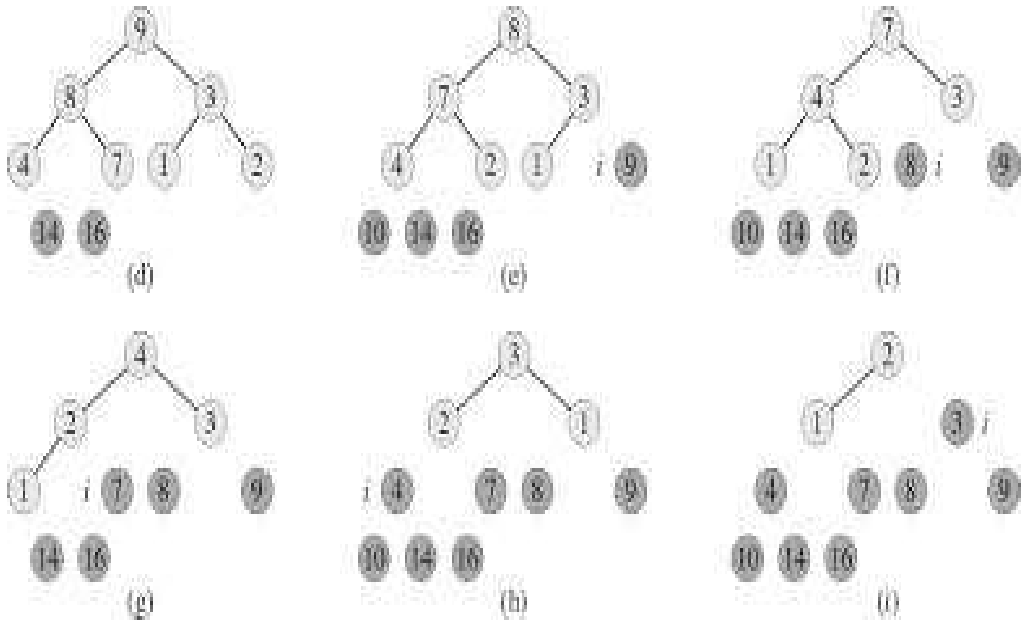
The total cost of BUILD-MAX-HEAP as being bounded is T(n)=O(n)

The HEAPSORT Algorithm

HEAPSORT(A)

1. BUILD MAX-HEAP(A)
2. for i=n to 2
3. exchange A[1] with A[i]
4. MAX-HEAPIFY(A,1)





A

1	2	3	4	7	8	9	10	14	16

TheHEAPSORT procedure takes time O(n log n), since the call to BUILD-MAX- HEAP takes time O(n) and each of the n - 1 calls to MAX-HEAPIFY takes time O(log n).

Lecture 10: Lower Bounds For Sorting

Review of Sorting: So far we have seen a number of algorithms for sorting a list of numbers in ascendingorder. Recall that an *in-place* sorting algorithm is one that uses no additional array storage (however,we allow Quicksort to be called in-place even though they need a stack of size $O(\log n)$ for keepingtrack of the recursion). A sorting algorithm is *stable* if duplicate elements remain in the same relativeposition after sorting.

Slow Algorithms: Include BubbleSort, InsertionSort, and SelectionSort. These are all simple $\Theta(n^2)$in-place sorting algorithms. BubbleSort and InsertionSort can be implemented as stable algorithms,but SelectionSort cannot (without significant modifications).

Mergesort: Mergesort is a stable $\Theta(n \log n)$ sorting algorithm. The downside is that MergeSort isthe only algorithm of the three that requires additional array storage, implying that it is not anin-place algorithm.

Quicksort: Widely regarded as the *fastest* of the fast algorithms. This algorithm is $O(n \log n)$ in the*expected case*, and $O(n2)$ in the worst case. The probability that the algorithm takes asymptoticallylonger (assuming that the pivot is chosen randomly) is extremely small for large n. It is an(almost) in-place sorting algorithm but is not stable.

Heapsort: Heapsort is based on a nice data structure, called a *heap*, which is a fast priority queue.Elements can be inserted into a heap in $O(\log n)$ time, and the largest item can be extracted in$O(\log n)$ time. (It is also easy to set up a heap for extracting the smallest item.) If you only wantto extract the k largest values, a heap can allow you to do this is $O(n + k \log n)$ time. It is anin-place algorithm, but it is not stable.

Lower Bounds for Comparison-Based Sorting: Can we sort faster than $O(n \log n)$ time?
We will give anargument that if the sorting algorithm is based solely on making comparisons between the keys in thearray, then it is impossible to sort more efficiently than $(n \log n)$ time. Such an algorithm is called a*comparison-based* sorting algorithm, and includes all of the algorithms given above.Virtually all known general purpose sorting algorithms are based on making comparisons, so this isnot a very restrictive assumption. This does not preclude the possibility of a sorting algorithm whoseactions are determined by other types of operations, for example, consulting the individual bits ofnumbers, performing arithmetic operations, indexing into an array based on arithmetic operations onkeys.We will show that any *comparison-based* sorting algorithm for a input sequence $ha1; a2; : : : ; ani$must
make at least $(n \log n)$ comparisons in the worst-case. This is still a difficult task if you think about it.It is easy to show that a problem *can* be solved fast (just give an algorithm). But to show that a problem*cannot* be solved fast you need to reason in some way about all the possible algorithms that might everbe written. In fact, it seems surprising that you could even hope to prove such a thing. The catch hereis that we are limited to using comparison-based algorithms, and there is a clean mathematical way ofcharacterizing all such algorithms.

Decision Tree Argument: In order to prove lower bounds, we need an abstract way of modeling "any possible"comparison-based sorting algorithm, we model such algorithms in terms of an abstract modelcalled a *decision tree*.In a *comparison-based* sorting algorithm only comparisons between the keys are used to determinethe action of the algorithm. Let *ha*1*; a*2*; : : : ; ani*be the input sequence. Given two elements, *ai*and*aj*, their relative order can only be determined by the results of comparisons like*ai*<*aj*, *ai*<=*aj*,*ai*=*aj*, *ai*>=*aj*, and *ai*>*aj*.A decision tree is a mathematical representation of a sorting algorithm (for a fixed value of *n*). Eachnode of the decision tree represents a comparison made in the algorithm (e.g., *a*4 : *a*7), and the twobranches represent the possible results, for example, the left subtree consists of the remaining comparisonsmade under the assumption that *a*4 _ *a*7 and the right subtree for *a*4 > *a*7. (Alternatively, onemight be labeled with *a*4 < *a*7 and the other with *a*4 _ *a*7.)Observe that once we know the value of *n*, then the "action" of the sorting algorithm is completelydetermined by the results of its comparisons. This action may involve moving elements around in thearray, copying them to other locations in memory, performing various arithmetic operations on non-keydata. But the bottom-line is that at the end of the algorithm, the keys will be permuted in the final array in some way. Each leaf in the decision tree is labeled with the final permutation that the algorithmgenerates after making all of its comparisons.To make this more concrete, let us assume that *n* = 3, and let's build a decision tree for SelectionSort.Recall that the algorithm consists of two phases. The first finds the smallest element of the entire list,and swaps it with the first element. The second finds the smaller of the remaining two items, and swapsit with the second element. Here is the decision tree (in outline form). The first comparison is between*a*1 and *a*2. The possible results are:

*a*1 <= *a*2**:** Then *a*1 is the current minimum. Next we compare *a*1 with *a*3 whose results might be either:

*a*1 <=*a*3**:** Then we know that *a*1 is the minimum overall, and the elements remain in their originalpositions. Then we pass to phase 2 and compare *a*2 with *a*3. The possible results are:

*a*2 <=*a*3**:** Final output is *ha*1*; a*2*; a*3*i*.

*a*2 > *a*3**:** These two are swapped and the final output is *ha*1*; a*3*; a*2*i*.

*a*1 > *a*3**:** Then we know that *a*3 is the minimum is the overall minimum, and it is swapped with*a*1. The we pass to phase 2 and compare *a*2 with *a*1 (which is now in the third position ofthe array) yielding either:

*a*2 <=*a*1**:** Final output is *ha*3*; a*2*; a*1*i*.

*a*2 > *a*1**:** These two are swapped and the final output is *ha*3*; a*1*; a*2*i*.

*a*1 > *a*2**:** Then *a*2 is the current minimum. Next we compare *a*2 with *a*3 whose results might be either:

*a*2 <=*a*3**:** Then we know that *a*2 is the minimum overall. We swap *a*2 with *a*1, and then pass to phase 2, and compare the remaining items *a*1 and *a*3. The possible results are:

*a*1 <=*a*3**:** Final output is *ha*2*; a*1*; a*3*i*.

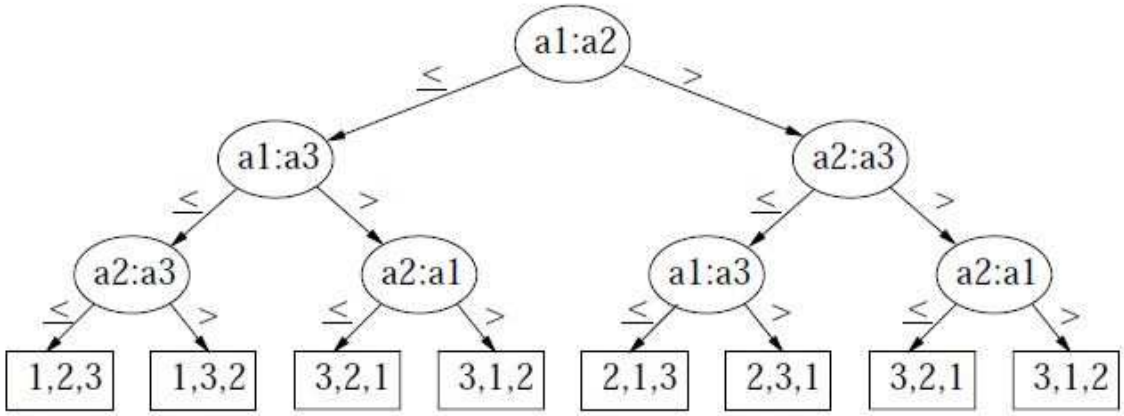
*a*1 > *a*3**:** These two are swapped and the final output is *ha*2*; a*3*; a*1*i*.

*a*2 > *a*3**:** Then we know that *a*3 is the minimum is the overall minimum, and it is swapped with*a*1. We pass to phase 2 and compare *a*2 with *a*1 (which is now in the third position of thearray) yielding either:

*a*2<= *a*1**:** Final output is *ha*3*; a*2*; a*1*i*.

*a*2 > *a*1**:** These two are swapped and the final output is *ha*3*; a*1*; a*2*i*.

The final decision tree is shown below. Note that there are some nodes that are unreachable. For example,in order to reach the fourth leaf from the left it must be that *a*1 _ *a*2 and *a*1 > *a*2, which cannotboth be true. Can you explain this? (The answer is that virtually all sorting algorithms, especiallyinefficient ones like selection sort, may make comparisons that are redundant, in the sense that theiroutcome has already been determined by earlier comparisons.) As you can see, converting a complexsorting algorithm like HeapSort into a decision tree for a large value of *n* will be very tedious andcomplex, but I hope you are convinced by this exercise that it can be done in a simple mechanical way.



(Decision Tree for SelectionSort on 3 keys.)

Using Decision Trees for Analyzing Sorting: Consider any sorting algorithm. Let $T(n)$ be the maximumnumber of comparisons that this algorithm makes on any input of size n. Notice that the running timefo the algorithm must be at least as large as $T(n)$, since we are not counting data movement or othercomputations at all. The algorithm defines a decision tree. Observe that the height of the decisiontree is exactly equal to $T(n)$, because any path from the root to a leaf corresponds to a sequence ofcomparisons made by the algorithm.

As we have seen earlier, any binary tree of height $T(n)$ has at most $2T(n)$ leaves. This means that thissorting algorithm can *distinguish* between at most $2T(n)$ different final actions. Let's call this quantity$A(n)$, for the number of different final actions the algorithm can take. Each action can be thought of asa specific way of permuting the original input to get the sorted output.How many possible actions must any sorting algorithm distinguish between? If the input consists of $n$distinct numbers, then those numbers could be presented in any of $n!$ different permutations. For eachdifferent permutation, the algorithm must "unscramble" the numbers in an essentially different way,that is it must take a different action, implying that $A(n) >= n!$. (Again, $A(n)$ is usually not exactlyequal to $n!$ because most algorithms contain some redundant unreachable leaves.)

Since $A(n) \leq 2T(n)$ we have $2T(n) \geq n!$, implying that

$T(n) \geq \lg(n!)$*:*

We can use *Stirling's approximation* for *n*! yielding:

$$n! \geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

$$T(n) \geq \log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)$$

$$= \log\sqrt{2\pi n} + n\log n - n\log e \in \Omega(n\log n)$$

Thus we have the following theorem.

Theorem: Any comparison-based sorting algorithm has worst-case running time ($n \log n$).
This can be generalized to show that the *average-case* time to sort is also ($n \log n$) (by arguing aboutthe average height of a leaf in a tree with at least *n*! leaves). The lower bound on sorting can begeneralized to provide lower bounds to a number of other problems as well.