

Module-I

Lecture 1 - Introduction to Design and analysis of algorithms

MOTIVATION

The advancement in science and technology enhance the performance of processor, which proportionally affect the characteristics of computer system, such as security, scalability and reusability. Important problems such as sorting, searching, string processing, graph problems, Combinational problems, numerical problems are basic motivations for designing algorithm.

DESIGN GOAL

The Basic objective of solving problem with multiple constraints such as problem size performance and cost in terms of space and time. The goal is to design fast, efficient and effective solution to a problem domain. Some problems are easy to solve and some are hard. Quite cleverness is required to design solution with fast and better approach. Designing new system need a new technology and background of the new technology is the enhancement of existing algorithm. The study of algorithm is to design efficient algorithm not only limited in reducing cost and time but to enhance scalability, reliability and availability.

The main concern of the course ensures:

- i) Correctness of solution
- ii) Decomposition of application into small and clear units which can be maintained precisely
- iii) Improving the performance of application

INTRODUCTION TO ANALYSIS OF ALGORITHM

A lay man perceives that a computer perform anything and everything. It is very difficult to ensure that it is not really the computer but the man behind computer who does the whole thing.

For example users just enter their queries and can get information as he/she desire. A common man rarely understands that a man made procedure called search has done the entire task and the only support provided by the computer is the execution speed and organized storage information.

‘Algorithm’ is defined after the name of **Abu Ja’ far Muhammad ibn Musa Al-Khwarizmi**, Ninth century , **al-jabr** means “restoring” referring to the process of moving a subtracted quantity to other side of an equation; **al-muqabala** is “comparing” and refers to subtracting equal quantities from both sides of an equation.

Definition of Algorithm

- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operations performed on data that have to be organized in the data structures.
- A finite set of instruction that specify a sequence of operations to be carried out in order to solve a specific problem or class of problems is called an algorithm.
- An algorithm is an abstraction of a program to be executed on a physical machine (model computation).

- An algorithm is defined as set of instructions to perform a specific task within finite no. of steps.
- Algorithm is defined as a step by step procedure to perform a specific task within finite number of steps.
- It can be defined as a sequence of definite and effective instructions, while terminates with the production of correct output from the given input.

Example of algorithm

Let us assume can common example of ATM withdrawal system .To present the scenario of a person under goes the following steps.

1. SWAP the card
2. Enter the PIN
3. Put the amount
4. Withdraw amount
5. Check balance
6. Cancel/clear.

If we go through these above 6 steps without considering the statement of the problem, we should assume that this is the algorithm for clearing the toilet. As of several ambiguities arises there while comprehending every step. The step 1 may imply toothbrush, paintbrush, toilet brush etc. Such an ambiguous doesn't an instruction an algorithmic step. Thus every step should be made unambiguous step is called '**definite instruction**'. Even if the step 2 is rewritten as apply the toothpaste, to eliminate ambiguities yet the conflicts such as, where to apply the toothpaste and where the source of the toothpaste is, need to be resolved. Therefore, the act of applying the toothpaste is not mentioned. Although unambiguous, such unrealizable steps can't be included as algorithmic instruction as they are not effective.

The definiteness and effectiveness of an instruction implies the successful termination of that instruction. Hence that above two may not be sufficient to guarantee the termination of the algorithm. Therefore, while designing an algorithm care should be taken to provide a proper termination for algorithm.

CHARACTERISTICS OF AN ALGORITHM

Every algorithm should have the following five characteristic features:

- a) Input
- b) Output
- c) Finiteness
- d) Definiteness
- e) Effectiveness
- f) Precision
- g) Determination
- h) Correctness
- i) Generality

a) Input

An algorithm may have one or more inputs. The inputs are taken from a specified set of subjects. The input pattern may be texts, images or any type of files.

b) Output

An algorithm may have one or more outputs. Output is basically a quantity which has a specified relation with the input.

c) Finiteness

An algorithm should terminate after a countable number of steps. In some cases the repetition of steps may be larger in number. If a procedure is able to resolve in finite number of execution of steps, then it is referred to be computational method.

d) Definiteness

Each step of an algorithm must be precisely defined. The action to be carried out must be unambiguously specified for each case. Due to the lack of understandability one may think that the step might be lacking definiteness. Therefore in such cases mathematical expressions are written, so that it resembles the instruction of any computer language.

e) Effectiveness

An algorithm is generally expected to be effective. Means the steps should be sufficiently basic, so that it may be possible for a man to resolve them.

f) Precision

The steps are precisely stated.

g) Determination

The intermediate results of each step of execution are unique and are determined only by input and output of the preceding steps.

h) Correctness

Output produced by the algorithm should be correct.

i) Generality

Algorithm applies to set of standard inputs.

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 algorithms run on computers or computational devices. for example, GPS in our smart phones, 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, value etc.)

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 250 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 Aposteriori 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

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

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

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 a sorting algorithm whose running time grows like $n \lg n$. They each must sort an array of 10 million numbers. Suppose that computer A executes 10 billion instructions per second (faster than any single sequential computer at the time of this writing) and computer B executes only 10 million instructions per second, so that computer A is 1000 times faster than computer B in raw computing power. To make the difference even more dramatic, suppose that the world's craftiest programmer codes in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers. Suppose further that 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 like $n \lg n$.

10 billion instructions per sec. 10 million instructions per sec

$2n^2$ instructions. $50n \lg n$ instructions.

Time taken = $\frac{2 \times 10^7 \times 10^7}{10^{10}} = 200$ $\frac{50 \times 10^7 \times \log 10^7}{10^7} \approx 1163$

It is more than 5.5 hrs 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 analyse 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 given below.

Pseudo code:

for $j=2$ to A length ----- C1

key=A[j]-----C2

//Insert A[j] into sorted Array A[1.....j-1]-----C3 $i=j-1$ -----
-----C4

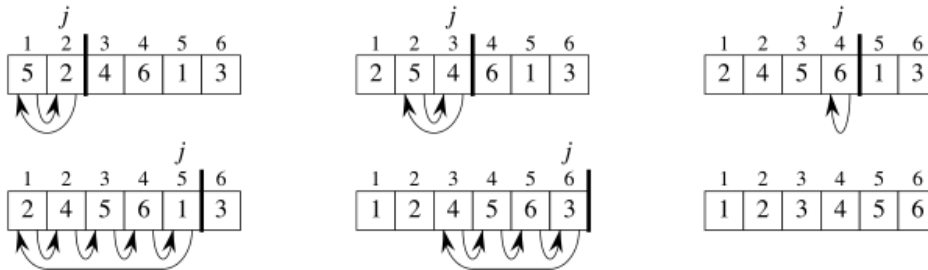
while $i>0$ & $A[j]>key$ -----C5

$A[i+1]=A[i]$ -----C6

$i=i-1$ -----C7

$A[i+1]=key$ -----C8

Example:



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

C_1n C_2n-1

$C_3=0$ $n-1$ C_4n-1 $C_5 \sum_{j=2}^{n-1} t_j$

$C_6 \sum_{j=2}^{n-1} (t_j - 1)$

$C_7 \sum_{j=2}^{n-1} (t_j - 1)$ C_8n-1

Running time of the algorithm is:

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

Best case:

It occurs when Array is sorted.

All devalues are 1.

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

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

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

Which is of the form $an+b$?

Linear function of n . so, linear growth.

Worst case: It occurs when Array is reverse sorted, and $t_j = j$

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

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

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 running time 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 n^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 growth is 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.

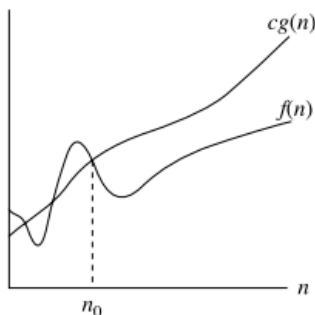
Focus on what's important by abstracting away low-order terms and constant factors.

It is a way to compare "sizes" of functions: $O \approx \leq$

$\Omega \approx \geq$ $\Theta \approx =$ $o \approx <$ $\omega \approx >$

***O*-notation**

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



$g(n)$ 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,

$$n$$

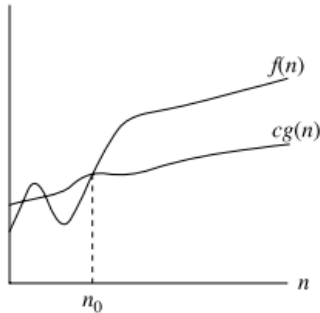
$$n/1000$$

$$n^{1.99999}$$

$$n^2 / \lg \lg \lg n$$

Ω -notation

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



$g(n)$ 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

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$
 $0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$.

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

***o*-notation**

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

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

$$n^{1.9999} = o(n^2)$$

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

$$n^2 \neq o(n^2) \text{ (just like } 2 \not\leq 2)$$

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

***ω*-notation**

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

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

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

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

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

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

$$T(n) = \Theta(1) \text{ ; if } n=1$$

$$2T(n/2) + \Theta(n) \text{ ; if } n>1 \text{ whose solution can be found as } T(n) = \Theta(n \log n)$$

There are various techniques to solve recurrences.

1. SUBSTITUTION METHOD:

The substitution method comprises of 3 steps

Guess the form of the solution

Verify by induction

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

$$\boxed{?} \quad F(n)=4f(n/2)$$

$$\boxed{?} \quad F(2n)=4f(n)$$

$$\boxed{?} \quad F(n)=n^2$$

So, $T(n)$ is order of n^2 Guess $T(n)=O(n^3)$

Step 2: verify the induction

Assume $T(k) \leq ck^3$ $T(n)=4T(n/2)+n$

$$\leq 4c(n/2)^3 + n$$

$$\leq cn^3/2 + n$$

$$\leq cn^3 - (cn^3/2 - n)$$

$T(n) \leq cn^3$ as $(cn^3/2 - n)$ is always positive So what we assumed was true.

$$\boxed{?} \quad T(n)=O(n^3)$$

Step 3: solve for constants $Cn^3/2 - n \geq 0$

$$\boxed{?} \quad n \geq 1$$

$$\boxed{?} \quad c \geq 2$$

Now suppose we guess that $T(n)=O(n^2)$ which is tight upper bound Assume $T(k) \leq ck^2$

so, we should prove that $T(n) \leq cn^2$

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

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

$$= cn^2 + n$$

So, $T(n)$ will never be less than cn^2 . But if we will take the assumption of $T(k)=c_1 k^2 - c_2 k$, then we can find that $T(n) = O(n^2)$

2. BY ITERATIVE METHOD:

$$\text{e.g. } T(n) = 2T(n/2) + n$$

$$\Rightarrow 2[2T(n/4) + n/2] + n$$

$$\Rightarrow 22T(n/4) + n + n$$

$$\Rightarrow 22[2T(n/8) + n/4] + 2n$$

$$\Rightarrow 23T(n/23) + 3n$$

After k iterations, $T(n) = 2^k T(n/2^k) + kn$ ----- (1) Sub problem size is 1 after $n/2^k = 1 \Rightarrow k = \log n$ So, after $\log n$ iterations, the sub-problem size will be 1. So, when $k = \log n$ is put in equation 1 $T(n) = nT(1) + n \log n$

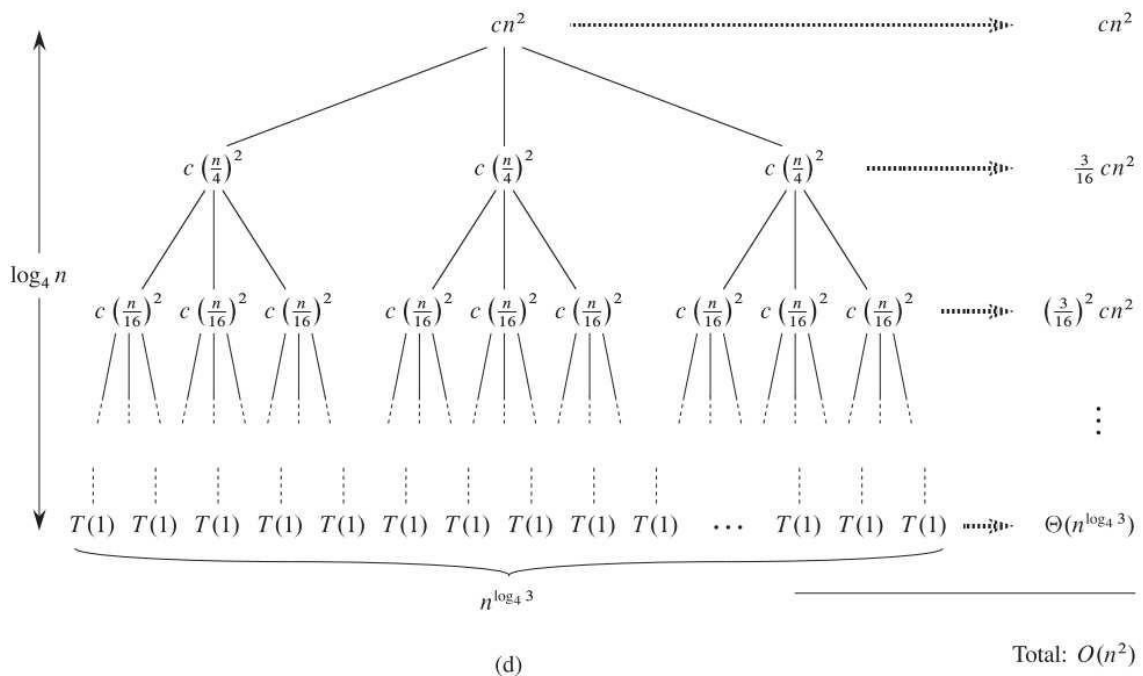
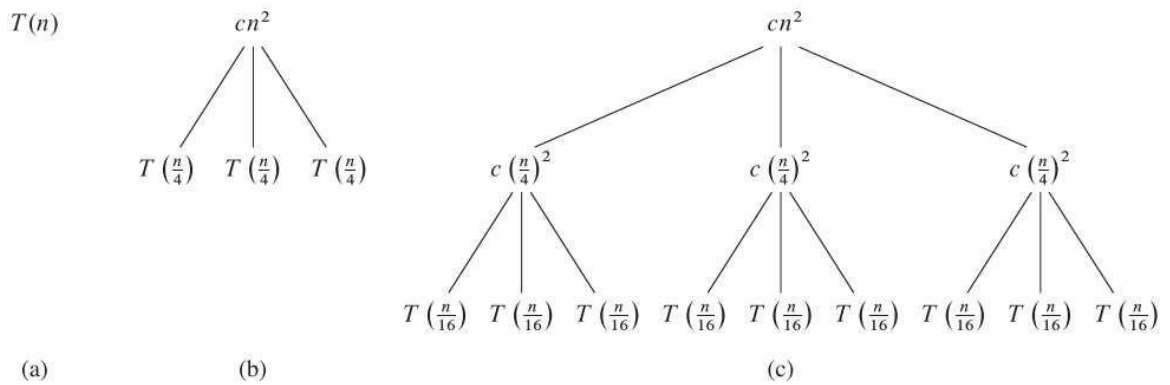
$$= nc + n \log n \quad (\text{say } c = T(1))$$

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

3. BY RECURSION 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 $n/4^i = 1 \Rightarrow i = \log_4 n$ So, no. of levels $= 1 + \log_4 n$

Cost of each level = (no. of nodes) \times (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 $i=3^i c(n/4^i)^2 = (3/16)^i cn^2$ $T(n) = \sum_{i=0}^{\log_4 n} cn^2(3/16)^i$

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

② $c3^{\log_4 n}$ (at last level $i=\log_4 n$)

② $cn \log_4 3$

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

$\leq cn^2$

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

② $\leq cn^2 * (16/13) + cn \log_4 3 \Rightarrow T(n) = O(n^2)$

4. BY MASTER METHOD:

The master method solves recurrences of the form

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

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is a asymptotically positive function . To use the master method, we have to remember 3 cases:

If $f(n) = O(n \log^k a - \epsilon)$ for some constants $\epsilon > 0$, then $T(n) = \Theta(n \log^k a)$

If $f(n) = \Theta(n \log^k a)$ then $T(n) = \Theta(n \log^k a \log n)$

If $f(n) = \Omega(n \log^k a + \epsilon)$ for some constant $\epsilon > 0$, and if $a * f(n/b) \leq c * f(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$

$$T(n) = 2T(n/2) + n \log n$$

ans: $a=2$ $b=2$

$$f(n) = n \log n$$

using 2nd formula $f(n) = \Theta(n \log^2 \log n)$

$$\Rightarrow \Theta(n^1 \log^2 n) = n \log n \quad \Rightarrow K=1$$

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

$$\Rightarrow \Theta(n \log^2 n)$$

Lecture 6 - 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 numbers which we will assume is stored in an array $A[1...n]$. The objective is to output a permutation of this sequence, sorted in increasing order. This is normally done by permuting the elements within the array A .

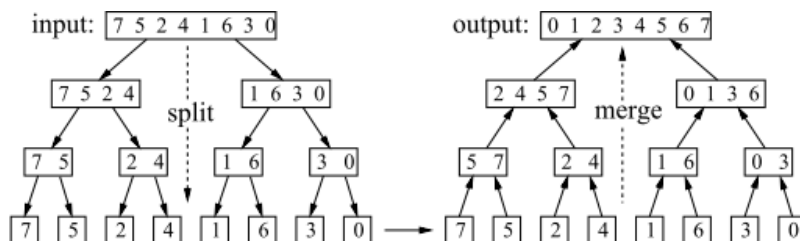
How can we apply divide-and-conquer to sorting? Here are the major elements of the Merge Sort algorithm.

Divide: Split A down the middle into two sub-sequences, each of size roughly $n/2$. Conquer: Sort each subsequence (by calling Merge Sort 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 subsists. The “conquer and combine” phases are shown 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 that merges two sorted list 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 last indices of the sub array that we are to sort. The call $\text{Merge Sort}(A, p, r)$ will sort the sub-array $[p..r]$ and return the sorted result in the same sub array.

Here is the overview. If $r = p$, then this means that there is only one element to sort, and we may return immediately. 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 the nearest integer). Then we split the array into sub arrays $A[p..q]$ and $A[q$

$+ 1 .. r]$. Call Merge Sort recursively to sort each sub array. Finally, we invoke a procedure (which we have yet to write) which merges these two sub arrays into a single sorted array.

Merge Sort(array A , into p , int r) {

if ($p < r$) { // we have at least 2 items $q = (p + r) / 2$

```

Merge Sort(A, p, q) // sort A[p..q]

Merge Sort(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 sub array, A [p..q], and the right sub array, A [q + 1 ..r], have already been sorted. We merge these two sub arrays by copying the elements to a temporary working array called B. For convenience, we will assume that the array B has the same index range A, that is, B [p..r]. We have two indices i and j, that point to the current elements of each sub array. We move the smaller element into the next position of B (indicated by index k) and then increment the corresponding index (either i or j). When we run out of elements in one array, then we 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 sub arrays are nonempty

if (A[i] <= A[j]) B[k++] = A[i++] // copy from left sub array

else B[k++] = A[j++] // copy from right sub array

}

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 Merge Sort. First let us consider the running time of the procedure Merge(A, p, q, r). Let $n = r - p + 1$ denote the total length of both the left and right sub arrays. What is the running time of Merge as a function of n? The algorithm contains four loops (none nested in the other). It is easy to see that each loop can be executed at most n times. (If you are a bit more careful you can actually see that all the while-loops

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

Now, how do we describe the running time of the entire Merge Sort algorithm? We will do this through the use of a recurrence, that is, a function that is defined recursively in terms of itself. To avoid circularity, the recurrence for a given value of n is defined in terms of values that are strictly smaller than 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 Merge Sort. Let $T(n)$ denote the worst case running time of Merge Sort on an 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 the Merge procedure.

First observe that if we call Merge Sort with a list containing a single element, then the running time is constant. Since we are ignoring constant factors, we can just write $T(n) = 1$. When we call Merge Sort with 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 sub array A [p..q], which contains $q - p + 1$ elements. You can verify that is of size $n/2$. Thus the remaining sub array A [q + 1 ..r] has $n/2$ elements in it. How long does it take to sort the left sub array? 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 sub array 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: $O(n^2)$.

Expected running time: $O(n \lg n)$.

Sorts in place.

Description of quick sort

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

To sort the subarea [p . . r]:

Divide: Partition A[p . . r], into two (possibly empty) subarrays A[p . . q - 1] and

A[q + 1 . . r], such that each element in the first subarray A[p . . q - 1] is $\leq A[q]$ and

A[q] is \leq each element in the second subarray A[q + 1 . . r].

Conquer: Sort the two sub arrays by recursive calls to QUICKSORT.

Combine: No work is needed to combine the sub arrays, 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 sub arrays.

QUICKSORT (A, p, r)

if $p < r$

then $q \leftarrow \text{PARTITION}(A, p, r)$

QUICKSORT (A, p, q - 1) QUICKSORT (A, q + 1, r)

Initial call is QUICKSORT (A, 1, n)

Partitioning

Partition subarray A [p . . r] by the following procedure: PARTITION (A, p, r)

$x \leftarrow A[r]$

$i \leftarrow p - 1$

for $j \leftarrow p$ to $r - 1$

do if $A[j] \leq x$, then $i \leftarrow i + 1$

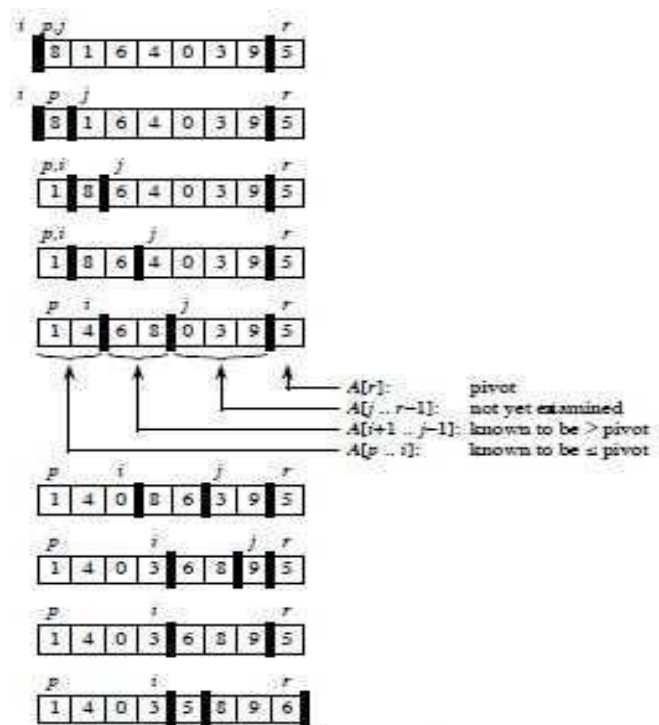
exchange $A[i] \leftrightarrow A[j]$

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 Quick sort

The running time of Quick sort depends on the partitioning of the sub arrays:

- If the sub arrays are balanced, then Quick sort can run as fast as merge sort.
- If they are unbalanced, then Quick sort can run as slowly as insertion sort.

Worst case

- Occurs when the sub arrays are completely unbalanced.
- Have 0 elements in one sub array and $n - 1$ elements in the other sub array.

- Get the recurrence

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

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

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

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

Best case

- Occurs when the sub arrays are completely balanced every time.
- Each sub array has $\leq n/2$ elements.
- Get the recurrence

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

Balanced partitioning

- QuickSort'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

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

- 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 $\log_{10} n$ full levels and $\log_{10/9} n$ 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 7 - 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 $T(n) = \begin{cases} \Theta(1), & \text{if } n \leq C \\ aT(n/b) + D(n) + C(n), & \text{otherwise} \end{cases}$
then n =input size a =no. Of sub-problems b = input size of the sub-problems

Lecture 8-Heaps and Heap sort

HEAPSORT

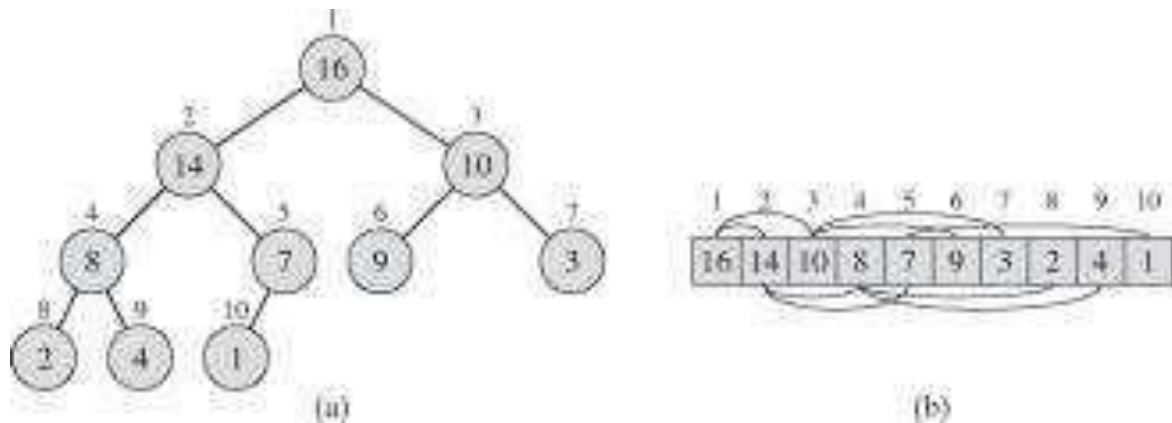
In place algorithm

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) \Rightarrow return $\lfloor i / 2 \rfloor$

LEFT (i) \Rightarrow return $2i$

RIGHT (i) \Rightarrow 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 $\lfloor i/2 \rfloor$ by shifting i right one bit position. Good implementations of heap sort 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, $A[\text{PARENT}(i)] \geq A[i]$, 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, $A[\text{PARENT}(i)] \leq A[i]$,

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)

$l \leftarrow \text{LEFT}(i)$

$r \leftarrow \text{RIGHT}(i)$

if $A[l] > A[i]$

largest $\leftarrow l$

if $A[r] > A[\text{largest}]$

largest $\leftarrow r$

if largest $\neq i$

Then exchange $A[i] \leftrightarrow A[\text{largest}]$

Lecture 9-MAX-HEAPIFY(A, largest)

At each step, the largest of the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$ 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 $A[\text{largest}]$, 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.

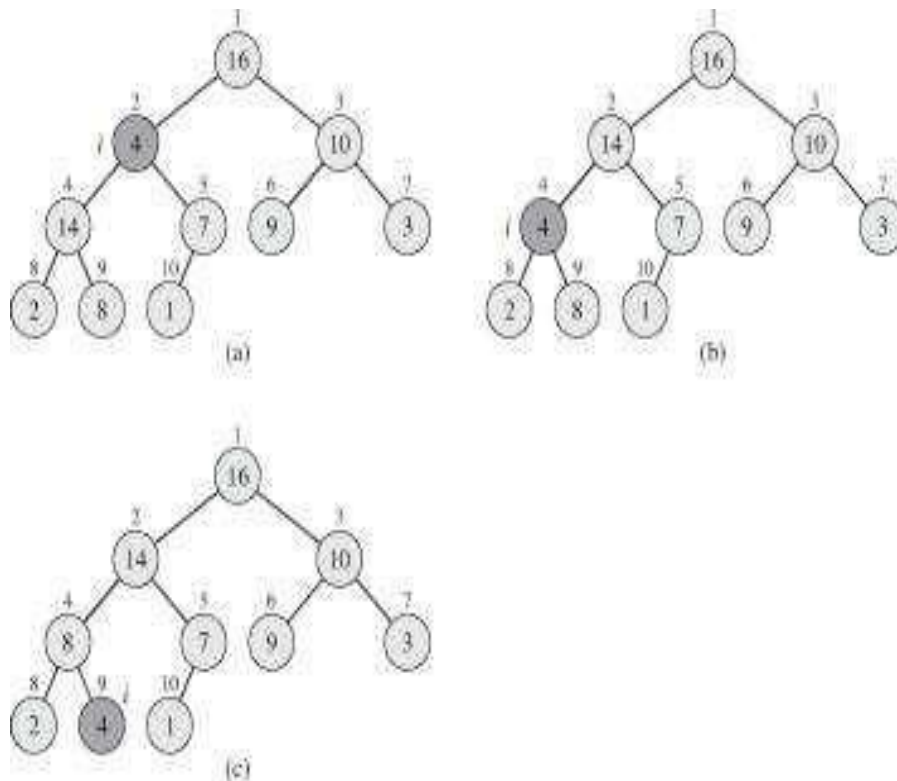


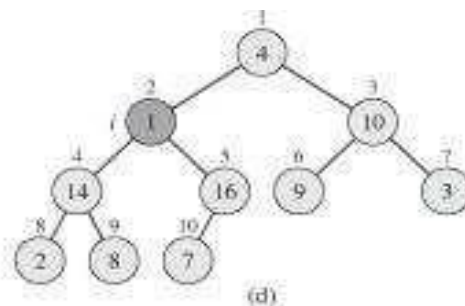
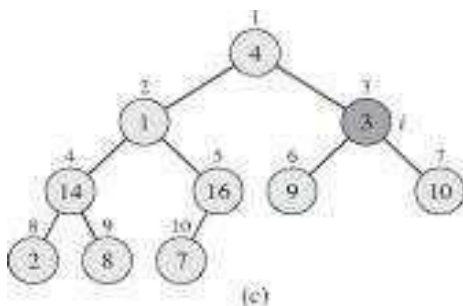
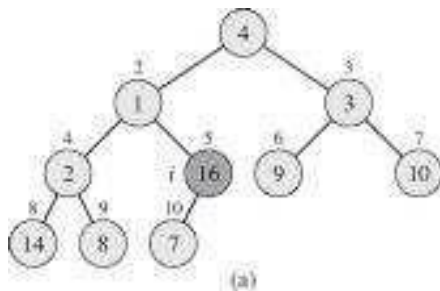
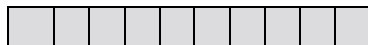
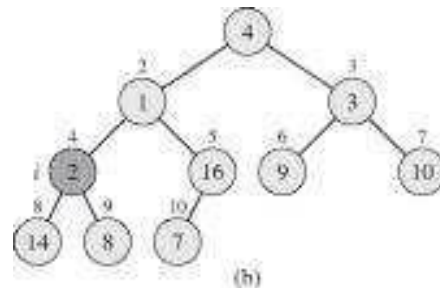
Figure: The action of MAX-HEAPIFY ($A, 2$), where heap-size = 10. (a) The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in (b) by exchanging $A[2]$ with $A[4]$, which destroys 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 $\text{MAX-HEAPIFY}(A, 9)$ yields no further change to the data structure. The running time of MAX-HEAPIFY by the recurrence can be described as $T(n) \leq T(2n/3) + O(1)$. The solution to this recurrence is $T(n) = O(\log n)$.

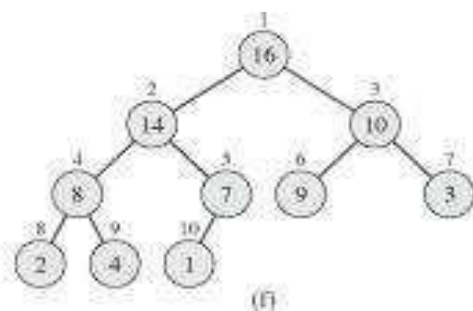
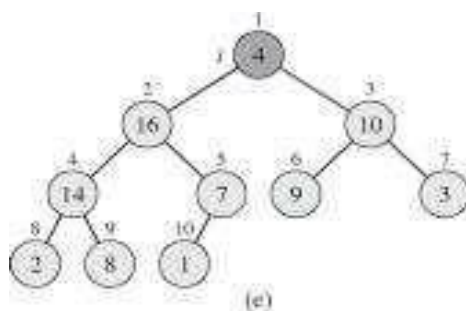
Building a heap

$\text{Build-Max-Heap}(A)$

for $i \leftarrow \lfloor n/2 \rfloor$ to 1
do $\text{MAX-HEAPIFY}(A, i)$



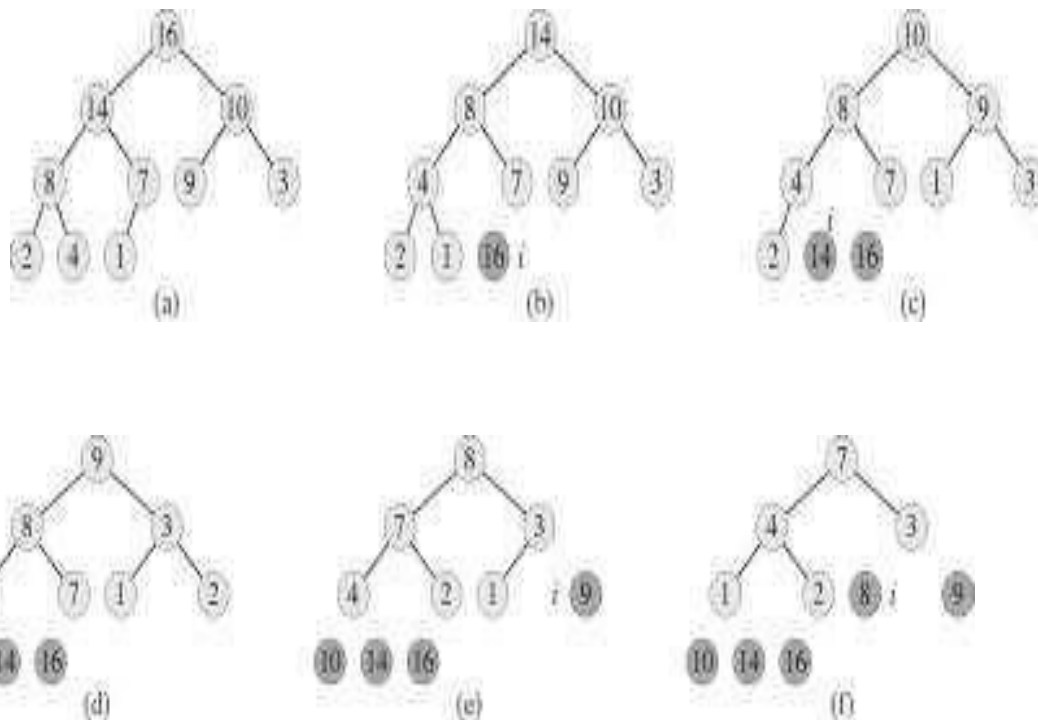
•



- 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 $\lceil \log n \rceil$ and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h .
- The total cost of BUILD-MAX-HEAP as being bounded is $T(n)=O(n)$

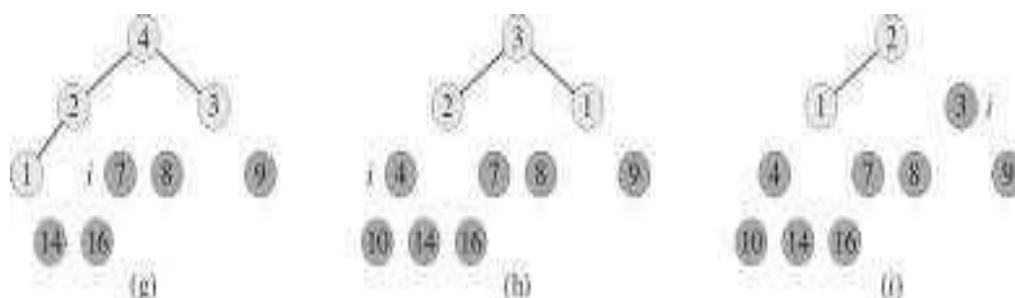
The HEAPSORT Algorithm

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



A

1	2	3	4	7	8	9	10	14	16
---	---	---	---	---	---	---	----	----	----



The HEAPSORT 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 ascending order. Recall that an in-place sorting algorithm is one that uses no additional array storage (however, we allow Quick sort to be called in-place even though they need a stack of size $O(\log n)$ for keeping track of the recursion). A sorting algorithm is stable if duplicate elements remain in the same relative position after sorting.

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

Merge sort: Merge sort is a stable $\Theta(n \log n)$ sorting algorithm. The downside is that Merge Sort is the only algorithm of the three that requires additional array storage, implying that it is not an on-place algorithm

Quick sort:

Widely regarded as the fastest of the fast algorithms. This algorithm is $O(n \log n)$ in the expected case, and $O(n^2)$ in the worst case. The probability that the algorithm takes asymptotically longer (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.

Heap sort: Heap sort 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 want to extract the k largest values, a heap can allow you to do this in $O(n + k \log n)$ time. It is a non-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 an argument that if the sorting algorithm is based solely on making comparisons between the keys in the array, then it is impossible to sort more efficiently than $(n \log n)$ time. Such an algorithm is called 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 is not a very restrictive assumption. This does not preclude the possibility of a sorting algorithm whose actions are determined by other types of operations, for example, consulting the individual bits of numbers, performing arithmetic operations, indexing into an array based on arithmetic operations on keys. We will show that any comparison-based sorting algorithm for a input sequence $a_1; a_2; \dots; a_n$ 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 ever be written. In fact, it seems surprising that you could even hope to prove such a thing. The catch here is that we are limited to using comparison-based algorithms, and there is a clean mathematical way of characterizing 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 model called a decision tree. In a comparison-based sorting algorithm only comparisons between the keys are used to determine the action of the algorithm. Let $a_1; a_2; \dots; a_n$ be the input sequence. Given two elements, a_i and a_j , their relative order can only be determined by the results of comparisons like $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, and $a_i > a_j$. A decision tree is a mathematical representation of a sorting algorithm (for a fixed value of n). Each node of the decision tree represents a comparison made in the algorithm (e.g., $a_4 > a_7$), and the two branches represent the possible results, for example, the left sub tree consists

of the remaining comparisons made under the assumption that $a_4 \leq a_7$ and the right sub tree for $a_4 > a_7$. (Alternatively, one might be labeled with $a_4 < a_7$ and the other with $a_4 \leq a_7$.) Observe that once we know the value of n , then the “action” of the sorting algorithm is completely determined by the results of its comparisons. This action may involve moving elements around in the array, copying them to other locations in memory, performing various arithmetic operations on non-key data. 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 algorithm generates 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 Selection Sort. 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 swaps it 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:

$a_1 \leq a_2$: Then a_1 is the current minimum. Next we compare a_1 with a_3 whose results might be either:

$a_1 \leq a_3$: Then we know that a_1 is the minimum overall, and the elements remain in their original positions. Then we pass to phase 2 and compare a_2 with a_3 . The possible results are:

$a_2 \leq a_3$: Final output is $ha_1; a_2; a_3$.

$a_2 > a_3$: These two are swapped and the final output is $ha_1; a_3; a_2$.

$a_1 > a_3$: Then we know that a_3 is the minimum is the overall minimum, and it is swapped with a_1 . Then we pass to phase 2 and compare a_2 with a_1 (which is now in the third position of the array) yielding either:

$a_2 \leq a_1$: Final output is $ha_3; a_2; a_1$.

$a_2 > a_1$: These two are swapped and the final output is $ha_3; a_1; a_2$.

$a_1 > a_2$: Then a_2 is the current minimum. Next we compare a_2 with a_3 whose results might be either:

$a_2 \leq a_3$: 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:

$a_1 \leq a_3$: Final output is $ha_2; a_1; a_3$.

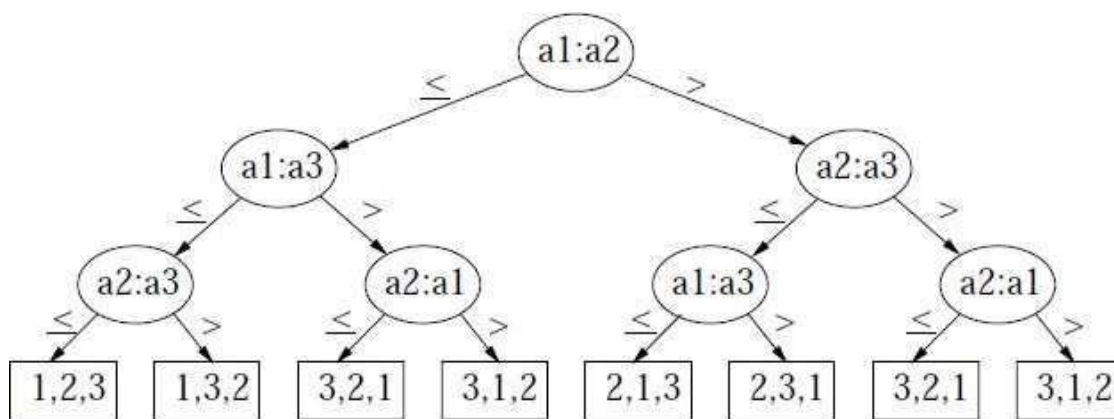
$a_1 > a_3$: These two are swapped and the final output is $ha_2; a_3; a_1$.

$a_2 > a_3$: 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 the array) yielding either:

$a_2 \leq a_1$: Final output is $ha_3; a_2; a_1$.

$a_2 > a_1$: These two are swapped and the final output is $ha_3; a_1; a_2$.

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 $a_1 \leq a_2$ and $a_1 > a_2$, which cannot both be true. Can you explain this? (The answer is that virtually all sorting algorithms, especially inefficient ones like selection sort, may make comparisons that are redundant, in the sense that their outcome has already been determined by earlier comparisons.) As you can see, converting a complex sorting algorithm like Heap Sort into a decision tree for a large value of n will be very tedious and complex, but I hope you are convinced by this exercise that it can be done in a simple mechanical way



(Decision Tree for Selection Sort on 3 keys.

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

As we have seen earlier, any binary tree of height $T(n)$ has at most $2^{T(n)}$ leaves. This means that this sorting algorithm can distinguish between at most $2^{T(n)}$ 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 as a 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 indistinct numbers, then those numbers could be presented in any of $n!$ different permutations. For each different permutation, the algorithm must "unscramble" the numbers in an essentially different way, that is it must take a different action, implying that $A(n) \geq n!$. (Again, $A(n)$ is usually not exactly equal to $n!$ because most algorithms contain some redundant unreachable leaves.)

- Since $A(n) \leq 2^{T(n)}$ we have $2^{T(n)} \geq n!$, implying that
- $T(n) \geq \lg(n!)$:

- We can use Sterling's approximation for $n!$ Yielding:

$$\begin{aligned}
 n! &\geq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \\
 T(n) &\geq \log \left(\sqrt{2\pi n} \left(\frac{n}{e}\right)^n \right) \\
 &= \log \sqrt{2\pi n} + n \log n - n \log e \quad \Omega(n \log n)
 \end{aligned}$$

- 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 about the average height of a leaf in a tree with at least $n!$ leaves). The lower bound on sorting can be generalized to provide lower bounds to a number of other problems as well.