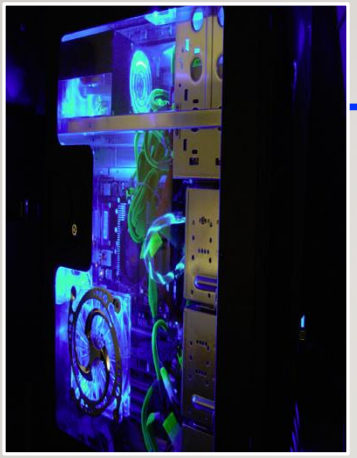
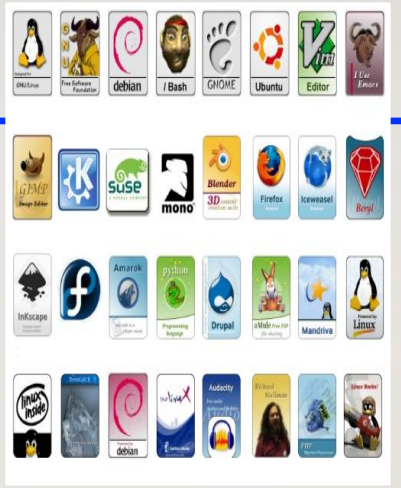
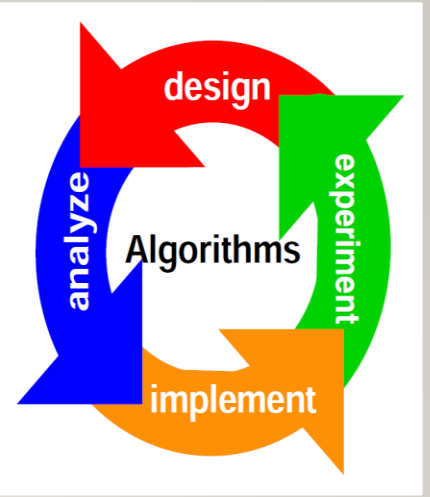
COMPUTER SCIENCE







Hardware

- Hardware
- Communication Infra-structure

Software

- Programs
- Software Infra-structure

Efficient algs+data str.

- Algorithms
- Data Structures

Introduction

"Algorithms change/d the world"

Precise instructions of how to add, subtract, multiply, divide, extract square roots, get digits of π,...

[Al Khwarizmi, AD 600, Baghdad]

Unambiguous, precise, mechanical, efficient and correct

Definition "Algorithm"

Informal:

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

An algorithm is thus a sequence of computational steps that transforms the input string into an output string.

Three questions to ask - algorithm

After we understand what it does or is supposed to do.

1. Is it correct?
2. How much time and/or space does it take, as a function of the input size?
3. Can we do better?

Correctness

Definition:

— An algorithm is correct if, for every input instance, it halts with the correct output.

— A correct algorithm solves the problem.

— An incorrect algorithm therefore:

- ... <see class>
- ... <see class>

Usefulness of incorrect algorithms

Sometimes, incorrect algorithms are useful or required

The algorithm

- must halt
- should give the correct result "mostly”
- Often algorithms may perform well in practice, but may fail sometimes.

Are they useful or desirable? Heuristics are examples. (see class for details)

Time and Space

We will measure time and space in "big oh”

Time:

— elementary operations count

Space Usage:

Counting units of

— Internal memory

Sometime, say in dbases, we count

— Disc -Tape

Establishing the Correctness of an Algorithm

By definition, it must: halt and return the correct result.

(1) We must show that it always halts; that part is usually easy.

(2) How to show that it returns the correct result?

That is sometimes interesting.

A Sequence

- The first 21 numbers are:

  0, 1,1, 2, 3, 5, *8,* 13, 21, 34, 55, 89, 144, 233

  377, 610, 987,1597, 2584, 4181, 6765

What can we say about this sequence?

Many things

- Non-negative
- Increasing
- Rapidly increasing almost as fast as 2^n, in fact a good approximation is: $2^{0.694n}$
- If we call F_n the n^{th} number then
  $F_n = F_{n-2} + F_{n-1}$ where $n>1$ and
  $F_0 = 0$ and $F_1 = 1$.

Fibonacci Numbers

SCIENCEphotOLIBRARY

F_n is called the n^{th} Fibonacci number.

Fibonacci was living between 1170 and 1250.

Fibonacci Numbers: brief sketch

Fibonacci numbers are also used for Fibonacci Search an alternate, O(log n) algorithm for searching in an array of n numbers.

Find smallest Fibonacci number, F_m >= n. Then split array in two of sizes F_{m-1} and F_{m-2}.

Recurse on the appropriate subarray after comparing your search key to the element in array position between the two subarrays.

Several advantages:

- It uses addition and subtract only as opposed to division by 2 as does binary search.
- It has a different array indexing pattern and caching etc. are different to binary search.
- If the array is too big to fit into main memory Fibonacci Search may outperform Binary Search,

Pineapple

The number of spirals going in each direction is a Fibonacci Number.

Here, there are 13 spirals that turn clockwise and 21 curving counterclockwise.

In sunflowers, the number of clockwise and counterclockwise spirals will always be consecutive Fibonacci Numbers like 21 and 34 or 55 and 34.

Fibonacci Occurrences in Nature

Number of Petals	Flower
3 petals (or 2 sets of 3)	lily (usually in 2 sets of 3 for 6 total), iris
5 petals	buttercup, wild rose, larkspur, columbine (aquilegia), vinca
8 petals	delphinium, coreopsis
13 petals	ragwort, marigold, cineraria
21 petals	aster, black-eyed susan, chicory
34 petals	plantain, daisy, pyrethrum
55 petals	daisy, the asteraceae family
89 petals	daisy, the asteraceae family

0, 1, *1, 2, 3, 5, 8, 13, 21,* 34, 55, 89, 144,...

Algorithm I for Fibonacci

Function fib1(n)

```
if n = 0 return 0
if n = 1 return 1
return fib1(n-1) + fib1(n-2)
```

The Three Questions

1. Is the algorithm correct?
2. How much time does it take to compute?
3. Can we do better?

1. is easy, that it follows simply by definition of the Fibonacci sequence. But let us do this formally. Observe first that it always halts. Why? -> class

1. Is the algorithm correct?

1. It is a recursive algorithm. So, a proof by induction seems the best way.

   First observe that the algorithm always halts.

   base cases(l):

   if n = 0 the algorithms returns 0 & stops by definition, $F_0 = 0$, so: correct

   if n=1 the algorithm also returns 1 & stops by definition $F_1 = 1$ so: correct

1. Is the algorithm correct? cont'd

1. Assume therefore now that n>1 and that the algorithm has correctly computed fib I(0),…, fib I(n-1).

   Then, the algorithm returns

   "return fib I(n-1) + fib I(n-2)"

   This recursively calls fib I(n-l) and fib I(n-2) and adds these two values. By induction hypothesis, fib I(n-l) and fib I(n-2) have been correctly computed.

   By definition of F_n, $F_n = F_{n-2} + F_{n-1}$ for n≥2. Thus, the algorithm is correct.

2. How much time does it take to compute?

- The time is a function of n, let us call it T(n)
- $T(n) \leqq 2$, for $n \leqq 1$ why?
- For larger values of n, i.e., $n \geq 2$

$$T(n) = T(n-1) + T(n-2) + 3$$

so, we can see that $T(n) \geq F(n)$

bad news?!

to compute T(200) > F(200) > 2^{138} steps would be required which is huge.

E.g., the Japanese Fugaku can do 442.01 petaFlop. To compute F(200) would longer than the earth is expected to live.

1 quadrillion calculations per second is a Peta Flop.

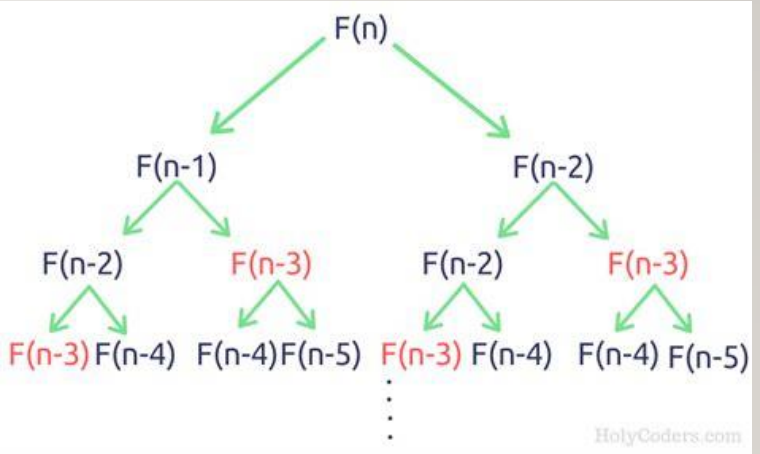
SO, CAN WE DO BETTER?

3. Can we do better?

Why is fib1 so bad?

Function fib1(n)

```
if n = 0 return 0
if n = 1return 1
return fib1(n-1) + fib1(n-2)
```



A SECOND ATTEMPT FIB2

Function fib2(n)

```
if n = 0 return 0
create an array f[0 .. n]
f[0] = 0, f[1] = 1
for i = 2 … n
        f[i] = f[i-1] + f[i-2]
return f[n]
```

Replace recursive calls
by array accesses.
Why recompute every time
when you can just do a look-up?

The three questions

1. Is the algorithm correct?
2. How much time does it take to compute?
3. Can we do better?

1. Is the algorithm correct?

   again, here the correctness (incl. that it halts) follows from the above as the recursive call is replaced by an equivalent array access.

2. How much time does it take to compute?

Except of the small number of constant-time operations the main work is carried out in the loop.

— the loop body is one addition

— and is executed n-1 times

So, the total number of addition etc. operations taken by fib2 is linear in n. Are all operations O(1)**?**

Careful

We get that fib2 executes a linear number of additions. However, the numbers are huge.

Our standard model of analysis assumes that the numbers have at most log n (minimum required to store the number n). [In real computers: constant number of bits (32 or 64)]. Here the numbers are about 0.694n bits long! Why? Such big numbers cannot be added in one step!

Adding 2 n-bit numbers takes time proportional to n. (bit operations) Such complications while rare, need to be carefully taken into consideration!

Conclusion

Lemma: The running time of fib2(n) is O(n) arithmetic operations, some of which take O(n) bit manipulations.

[This is more detailed than we usually argue in our "O" analysis because here, we have words which are longer than O(log n) bits. Rare though!]

What is the running time to compute the Fibonacci then?

- There is a difference in required run-time between computing the n^{th} Fibonacci number only, or all Fibonacci numbers up to the n^{th}.

3. Can we do better? YES, by reduction to fast matrix multiplication (not discussed here) the n^{th} Fibonacci number can be computed using O(log n) arithmetic operations.

Relationship between matrix multiplication and Fib – only for those interested

Very brief sketch of idea.

$$\begin{bmatrix} F(n+1) \\ F(n) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} F(n-1) \\ F(n-2) \end{bmatrix}$$

$$\cdots\cdots$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} F(1) \\ F(0) \end{bmatrix}$$

Doubling n is easily achieved by plugging in 2n instead of n. This leads to a logarithmic number of operations to compute:

$$\begin{bmatrix} F(2n+1) \\ F(2n) \end{bmatrix} = \begin{bmatrix} F(n+1)^2 + F(n)^2 \\ 2F(n+1)F(n) - F(n)^2 \end{bmatrix}$$

$$\begin{bmatrix} F(n+1) \\ F(n) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} F(n-1) \\ F(n-2) \end{bmatrix}$$

$$\cdots\cdots$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} F(1) \\ F(0) \end{bmatrix}$$

Golden ratio

- Two numbers a,b are said to be in the "golden ratio, Φ," if

$$\frac{a+b}{a} = \frac{a}{b} = \Phi$$

Golden ratio (equivalent)

- Two numbers a,b are in the "golden ratio, Φ," if

$$\frac{a+b}{a} = 1 + \frac{b}{a}$$

So, since b/a=1/Φ we get: 1+ 1/Φ= Φ

GOLDEN RATIO

Now solve $1 + 1/\Phi = \Phi$

$\Phi + 1 = \Phi^2$ rewritten as: $\Phi^2 - \Phi - 1 = 0$

By using the solutions to a quadratic formula,

$$ax^2 + bx + c \qquad x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

For $a = 1$, $b = -1$ and $c = -1$

we get:

$$\Phi_1 = (1+\sqrt{5})/2 \text{ and } \Phi_2 = (1-\sqrt{5})/2$$

Because Φ is a ration between positive quantities, Φ is positive. $\Phi = 1.618033988\ldots$

GOLDEN RATIO AND FIBONACCI

- By induction, we can prove: $F_i = (\Phi_1^i - \Phi_2^i)/\sqrt{5}$

- Note that $|\Phi_2^i|/\sqrt{5} < 1/\sqrt{5}$ (as $|\Phi_2|<1$) thus $|\Phi_2^i|/\sqrt{5} < ½$ and therefore:
- $F_i = \text{floor}(\Phi_1^i/\sqrt{5} + ½)$

- So: the i^{th} Fibonacci number is $\Phi_1^i/\sqrt{5}$ rounded to the nearest integer. Exponential growth!
- Furthermore, the ratio between two consecutive Fibonacci numbers approaches the Golden Ratio.

Recall Big-0

Let f(n) and g(n) be functions from the positive integers to the reals. We say that
f(n) = O(g(n)) if there is a constant c>0 such that

f(n) < c*g(n).

This means that f grows no faster than g.

Also, the constant c allows us to ignore small values of n.

Course requirement

I will assume that you are familiar with big-O.

if not, please ! please read up fast.

A COMMON STRATEGY: LOOP INVARIANT

- For algorithms that execute, e.g., a main loop (or a recurrence)
- Set up a **Loop invariant,** say L
- Say the algorithm executes a loop n times.
- **L(i):** statement about the algorithm is true before the i^{th} execution of the loop (i>0).
- **Base Case**: this is the base case, i.e., after initialization of the variables (if any), before entering the loop the first time. We need to show that for invariant is true for the base case.

Loop invariant continued

Termination: when the loop terminates, the truth of the invariant helps us establish the correctness of the algorithm.

Maintenance: prove that if L(i-1) is correct before the execution of the loop then L(i) is also true after the loop execution.

A VERY SIMPLE EXAMPLE

- A warm-up done in class
- Finding the maximum in an array A[1 ….n] of integers.

```
Max:= A[1]
For i = 2, …, n do
    if A[i]> Max then Max := A[i]
```

Discussion on this in-class

Insertion-Sort

```
INSERTION-SORT(A)
j:=2 // to make life easier to prove correctness //
for j:=2 to length of A
    key := A[j]
    // insert A[j] into A[1..j-1] //
    i:=j-1
    while (i>0 and A[i] > key)
        A[i+1] := A[i]
        i:=i -1
    A[i+1] := key
```

Example

- In class

Halting

The algorithm halts because it executes a finite number of statements a finite number of times, where each statement takes finite time to execute.

How do we proceed?

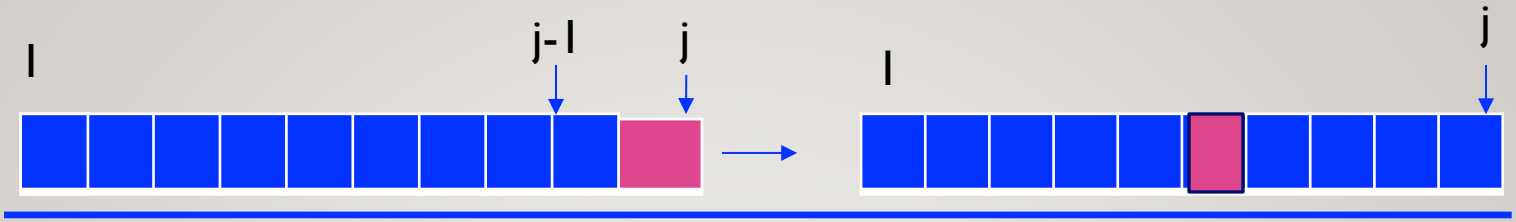
First, we will establish the correctness of the while-loop.

Then, the correctness of the For-loop

This will complete the entire proof.

Notation: A[i..j] is a subarray of array A consisting of the elements A[i], A[i+1],..., A[j]

CORRECTNESS OF THE WHILE-LOOP



Input: A[1..j-1] a sorted subarray of numbers and A[j] possibly in the wrong relative order w.r.t. A[1…j-1]

Output: A sorted array, A[1..j], of the numbers originally stored in A[1..j]

```
key := A[j]
i:=j-1
  while ( i>0 and A[i] > key)
        A[i+1] := A[i]     [move A[i] over]
        i:=i – 1           [proceed to the next element left]
   A[i+1] := key [place the key in its correct position
```

PROVING THE CORRECTNESS OF THE WHILE-LOOP

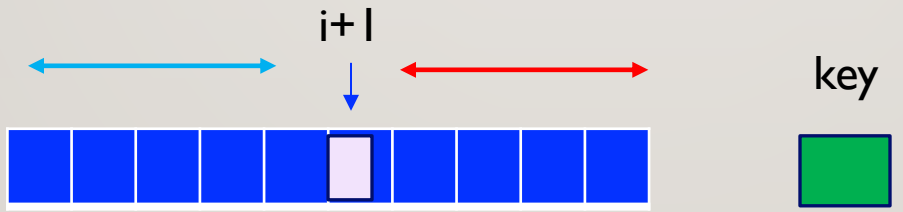
LOOP INVARIANT:

at the beginning of each iteration of the while-loop

A[i+2],…,A[j] are each greater than key and

A[i+2..j] contains the sorted values originally stored in A[i+1..j-1]

A[1..i] is untouched by the while loop



CORRECTNESS WHILE-LOOP CONT'D

Base-case

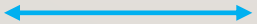
"at the beginning of each iteration of the while-loop

A[i+2],…,A[j] are each greater than key and

A[i+2..j] contains the sorted values originally stored in A[i+1..j-1]

A[1..i] is untouched by the while loop"

Before the first time through the while loop, i:=j-1
thus i+2 = j+1 and since A[j+1] does not exist the statements (A and B) are true.



Also (C) is true (A[1..i] is untouched) since we did enter the loop so far.

CORRECTNESS WHILE-LOOP CONT'D

Maintenance

"at the beginning of each iteration of the while-loop

A[i+2],…,A[j] are each greater than key and

A[i+2..j] contains the sorted values originally stored in A[i+1..j-1]

A[1..i] is untouched by the while loop"

Assume that the loop invariant is true and now we enter the while-loop.

Since A[i]>key we move A[i] to A[i+1] so now A[i+1]> key

After we decrement i, now A[i+2] >key (in addition to the other elements previously considered up to ….A[j])

The subarray of considered elements remains sorted

Subarray A[1 ..i] remains untouched and has shrunk by 1.

Correctness while-loop cont'd

Termination

"at the end of final iteration of the while-loop A[i+2],...,A[j] are each greater than key and A[i+2..j] are the sorted values originally stored in A[i+1..j-l]

A[1..i] is untouched by the algorithm"

Two cases arise:

1) The loop condition is violated because i = 0 or
2) A[i] ≤ key

Assume 1) i.e., i=0 then since A[i+2 ... j] is the sorted array of values originally stored in A[i+1.. j-1], for i=0, this means A[2..j] is the sorted array of values from A[1...j-1] and A[2],...A[j] are all greater than key. The final assignment statement A[i+1]=key puts A[1]=key thus A[1...j] is the sorted array of all values originally stored in A[1...j].

Correctness while-loop cont'd

Termination

"at the end of final iteration of the while-loop A[i+2],...,A[j] are each greater than key and A[i+2..j] are the sorted values originally stored in A[i+l..j-1]

A[1..i] is untouched by the algorithm"

Now assume 2), i.e., A[i] ≤ key

By loop-invariant, A[i+2 ... j] is the sorted array of the values originally stored in A[i+1.. j-1]; all elements are > key.

A[1..i] is untouched since A[1 ...i] was sorted before and A[i] ≤ key, by transitivity, all elements in A[1...i] are sorted and ≤ key.

The final assignment puts key into its correct position, i.e., A[i+1] = key thus the entire array A[1..j] is now sorted.

Setting up the invariant for Insertion Sort

- INVARIANT L[j]

At the start of each iteration of the **for** loop the subarray A[1...j-1] consists of the elements originally stored in A[1..j-1] but now sorted

(subarray A[j… A.length] is untouched – we will not prove this easy fact)

Correctness of Insertion Sort

Base Case:

Before the algorithm enters the loop,

what do we know?

We know that

- the input is stored an array
- j = 2

Base case cont'd

"At the start of each iteration of the **for** loop the subarray A[1...j-1] consists of the elements originally stored in A[1..j-1] but sorted."

Since j=2, we need to show that

"At the start of each iteration of the **for** loop the subarray A[1…1] consists of the elements originally stored in A[1..1] but now sorted."

A[1..1]=A[1].

- A[1] contains the element originally in A[1]
- A[1] by itself is clearly sorted

Termination

At **termination:**

If we have shown that L(j) is true then

"At the start of each iteration of the **for** loop the subarray A[1...j-1] consists of the elements originally stored in A[1..j-1], but now sorted."

when the loop terminates j > A.length, in fact j=A.length+1 as the loop-variable, j, is incremented by one each time.

So, the subarray A[1..A.length] is sorted and contains the elements originally stored in A. Thus, the algorithm returns the correct result.

Loop maintenance

At the start of each iteration of the **for-**loop, the subarray A[1...j-1] consists of the elements originally stored in A[1..j-1] but now sorted.

Now, consider j to be incremented

Key := A[j]

Key and A[1..j-1] contain all elements of A[1..j]

Loop maintenance cont'd

i=j-1 // sets the limit of the subarray to be examined subsequently//

```
    while (i>0 and A[i] > key)
        A[i+i]
        :=A[i] i:=i -
        1
    A[i+1] := key
```

We have already shown that if A[1..i] is a sorted array and key is a value then after the execution of the while-loop A[1..i+1] is a sorted array containing the values of A[1..i] and key.

Loop maintenance cont'd

i=j-1 // sets the limit of the subarray to be examined subsequently//

```
    while (i>0 and A[i] > key)
        A[i+i] :=A[i]
        i:=i -1
    A[i+1] := key
```

Since i=j-1, by loop-invariant, we have that A[1..j-1] is a sorted array of the values originally stored in A[1..j-1]. Key = A[j] so, with the above, A[1..j] will be a sorted array of the values originally stored in A[1..j]

2. Time and Space Complexity

The algorithm executes two nested loops, each loop is executed at most O(n) times. The inner loop has O(1) time complexity. Except for the inner loop, the outer loop is O(1).

Therefore, the total time complexity is $O(n^2)$.

Except for a constant number of additional storage locations, the only storage used is the input array; **the algorithm is "in-place"**.

Thus, the space complexity is O(n).

3. Can we do better?

Yes, we already know that e.g., MergeSort is an O(n log n) algorithm using O(n) space.

I recall the algorithm and its analysis briefly as another example of how to prove correctness.

MergeSort

```
MERGE-SORT(A, p, r)
// sorts a subarray A[p..r] of numbers//
    If p<r
        q = floor{(p+r)/2}
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```

Algorithm MERGE

MERGE(A,p,q,r)

//Forms a sorted (sub)array A[p..r] from two sorted (sub)arrays A[p..q] and A[q+1..r]; L, R are two temporary arrays//

n_1=p-q+1; n_2=r-q //sizes of the "input arrays"//

Copy A[p..q] into L[1.. n_1]; add L[n_1+1]=∞

Copy A[q+1..r] into R[1.. n_2]; add R[n_2+1]=∞

i=1; j =1

Algorithm MERGE cont'd

```
                    MERGE(A,p,q,r)
For k= p to r      //go through all elements //
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i+ l
    else A[k] = R[k]
        j = j+ l
```

First we analyze MERGE

1. **Correctness**

   Good exercise (when you are done then look at the textbook for a solution to verify yours against)

2. **Time Complexity**

   Let n= r-p+1

   MERGE has three loops examining O(n) elements

   the first two copy array elements into L and R, resp.

   the last executes the loop r-p+1 = O(n) times and performs O(1) operations at a cost of O(1) each; so

   total: O(n)

First we analyze MERGE cont'd

2. Space Complexity

   there are three arrays of size O(n)

3. Can we do better?

   no – we need to examine all elements thus Ω(n) is a lower bound.

(Lower bound means we cannot do better, i.e., no algorithm can be designed which performs the task in a smaller time bound. We must look at least at all elements ones.)

Same for space (we must store the elements as input; the additional arrays are all of the same size which does not change the space complexity in "O").

Question

- What would happen if we were to attempt the MERGE procedure in the original array A?
- Any impact on the complexity? Important exercise.

Divide&Conquer Algorithm D&C

- Recall D&C
- Break problem into sub-problems that are smaller instances of the same type of problem
- Recursively solve the subproblems
- Combine the answers to the subproblems into an answer to the original problem

Divide&Conquer Algorithm D&C

- illustration see class

Here: D&C MERGE-SORT

- Break problem into subproblems that are smaller instances of the same type of problem
  - "the subarrays: A[p..q] and A[q+1..r]"
- Recursively solve the subproblems
  - "MERGE-SORT(A,p,q) and MERGE-SORT(A,q+1,r)"
- Combine the answers to the subproblems into an answer to the original problem
  - "MERGE(A,p,q,r)"

1. Correctness of MERGE-SORT

There is no loop, but a straight recurrence. A proof by induction seems feasible.

Induction on the size of the array n=r-p+1.

Base Case: n=1 (i.e., r=p)

a single element array is sorted

Induction Hypothesis: assume that MERGE-SORT sorts any array of sizes 1,..., n-1, for n ≧2.

1. Correctness of MERGE-SORT cont'd

Show that it sorts any input array of size n

Since n ≧ 2, p<r thus the statements inside the "if-statement" are executed

With q, the array is split into two (roughly equal-sized) subarrays (A[p..q] and A[q+l..r])

Since their sizes are less than n, by induction and after calls to MERGE-SORT on them, they are correctly sorted.

The final step MERGE then sorts the two sorted subarrays (the correctness follows from the correctness of MERGE)

2. Complexity

Again let n = r-p+1

The space used is: O(n) because MERGE-SORT uses only a constant additional locations (note that the maintenance of the recurrence also takes some storage which is linear in n. This requires re-using of the additional array space used by MERGE! Otherwise, this would be O(n log n) !) MERGE has O(n) space usage as we said

2. Complexity cont'd

- Now, let us analyze the time complexity.
- It is given by this recurrence relation
- $T(n) = 2*T(n/2) + 0(n)$ with $T(1) = O(1)$
  — Why? See class
- There are many ways to solve this recurrence.
- You will have seen some already before. I recall.

Solving the recurrence

$T(n) = 2\,T(n/2) + O(n)$

a) By developing the recurrence:

Let us say it is exactly n not O(n) - makes notation easier

Otherwise, we can carry the constants through.

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

$=2^2T(n/2^2) + 2*n = 2^2[2T(n/2^3)+n/2^2)] + 2n$

$=2^3T(n/2^3) + 3*n = ...$

$= 2^iT(n/2^i) + i*n$, for all i=0,...,?

What is i maximally?

Solving recurrences cont'd

- i is maximally $\log_2 n$ because then $2^i = n$

$T(n) = 2^i T(n/2^i) + i*n = 2^{\log_2 n} T(n/2^{\log_2 n}) + (\log_2 n)*n$ since

$T(n/2^{\log_2 n}) = T(1) = O(1)$; $T(n) = O(n) + (\log_2 n)*n$

$=O(n)+n*\log_2 n = O(n*\log_2 n)$. We therefore obtain:

Lemma: MERGE-SORT sorts n elements in $O(n \log n)$ time.

3. Can we do better?

In our model of computation we count comparisons.

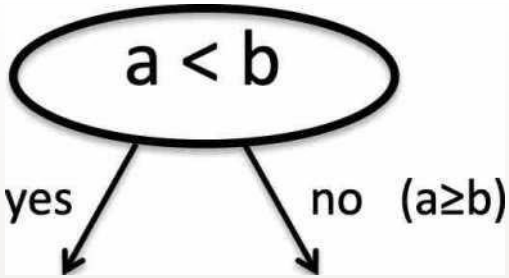
(In any sorting algorithm discussed, we compared elements to each other and then rearranged the array. So, this is natural to do. Not in all sorting algorithms mind you.)

Show: no sorting algorithm can sort n arbitrary numbers in o(n log n) comparisons.

(this is little "oh"; informally, that means with fewer than n log n comparisons)

Comparisons

A comparison between two numbers a, b



Consider now sorting three numbers a,b,c

Sorting by comparisons

In class:

comparison tree for sorting 3 elements

Sorting n elements counting comparisons only

Any comparison-based algorithm that sorts n elements has an execution that can be described by a comparison tree.

If we sort n elements, any permutation of the n elements can appear as output thus as one of the leaves of the tree.

Thus the tree must have n! leaves.

Height and Depth

*The **depth** of a node is the number of edges present in path from the root node of a **tree** to that node.*

*The **height** of a node is the number of edges present in the longest path connecting that node to a leaf node.*

Sorting n elements counting comparisons only - depth of tree

- The tree is binary.
- What is the smallest depth of a binary tree?

in-class A bin. tree with m leaves has depth of at least log m. (Omitting the ceiling function)

Thus, the depth > log(n!)

log(n!) > c*n log n, for some c>0

Why? *In-class*

a better bound for n!

Stirling's approximation provides a more precise bound

Stirling's formula: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Cont'd

- The depth of the tree corresponds to the number of operation (of type comparison) we need to do.

Theorem: Any comparison-based sorting algorithm has an $\Omega(n \log n)$ lower bound to sort n elements.

Corollary: MERGE-SORT is thus an optimal comparison-based algorithm.

We need not look any further to get a better algorithm (in terms of "big oh").

Back to Recurrences

What is the complexity of binary search?

Brief recall in-class

2. Time complexity

Recurrence: Let n be the size of the array in which we carry out binary search.

$T(n) = T(n/2) + O(1)$

$T(n) = T(n/2^1) + 1*O(1) = T(n/4) + 2*O(1)$

$= T(n/2^2) + 2*O(1)$

$= T(n/8) + O(1) + O(1) + O(1)$

$= T(n/2^3) + 3*O(1)$... by induction

$= T(n/2^i) + i*O(1)$ for $i = 1, \ldots, \log_2 n$

$= T(n/2^{\log_2 n}) + \log_2 n * O(1) = O(\log_2 n)$

as $T(1) = O(1)$

Another example

(arises in the analysis of a multiplication algorithm - not important right now here) $T(n) = 3\,T(n/2) + n$

$= 3[3T(n/2^2) + n/2^1] + n$

$= 3^2T(n/2^2) + 3n/2^1 + n$ =... <by induction>

$= {}_3{}^{\log_2 n}O(1) + .. + (3/2)^i n + ... + (3/2)^1 n + n$

Note: ${}_3{}^{\log_2 n} = n^{\log_2 3}$

This is a geometric series -> $0(n^{\log_2 3}) = 0(n^{1.59})$

Rules for working with logarithms

Rule name	Rule
product rule	$\log_b(x \cdot y) = \log_b(x) + \log_b(y)$
quotient rule	$\log_b(x / y) = \log_b(x) - \log_b(y)$
power rule	$\log_b(x^y) = y \cdot \log_b(x)$
base switch rule	$\log_b(c) = 1 / \log_c(b)$
base change rule	$\log_b(x) = \log_c(x) / \log_c(b)$

Why is $3^{\log_2 n} = n^{\log_2 3}$?

- $3^{\log_2 n} = n^{\log_2 3}$
- Take the log base n on both sides:
- Obtain: $\log_2 n \log_n 3 = \log_2 3$
- Divide both sides by $\log_2 n$
- Obtain $\log_n 3 = \log_2 3 / \log_2 n$

- **base change rule** $\log_b(x) = \log_c(x) / \log_c(b)$
  - With b= n , x = 3 and c=2 the above is thus correct

Geometric Series

- For $x \neq 1$, the summation $\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n$ is a geometric series and has value:

$$= \frac{x^{n+1} - 1}{x - 1}$$

- When $|x| < 1$, we get an infinite decreasing geometric series $\sum_{k=0}^{\infty} x^k$ converging to $1/(1-x)$.

Solving Recurrences by Substitution

The previous method, also known as Iteration Method, often works, but is sometime error-prone.

If you have a good guess of what the solution is then, the following method is faster to use.

Substitution Method

1. Make a guess
2. Prove using induction

Example: $T(n) = T(n-1) + n$

1. Guess $T(n) \leq cn^2$
2. Prove using induction

Assume that $T(k) \leq ck^2$ for $k < n$, for some constant $c>1$

$T(n) = T(n-1) + n \leq c(n-1)^2 + n \leq cn^2 - 2cn + c + n$

$= cn^2 - c(2n-1) + n \leq cn^2$ for $c>1$ as $-c(2n-1) < n$

(actually $c>1/2$ would do)

Recall Tutorial

Recursion Tree Method

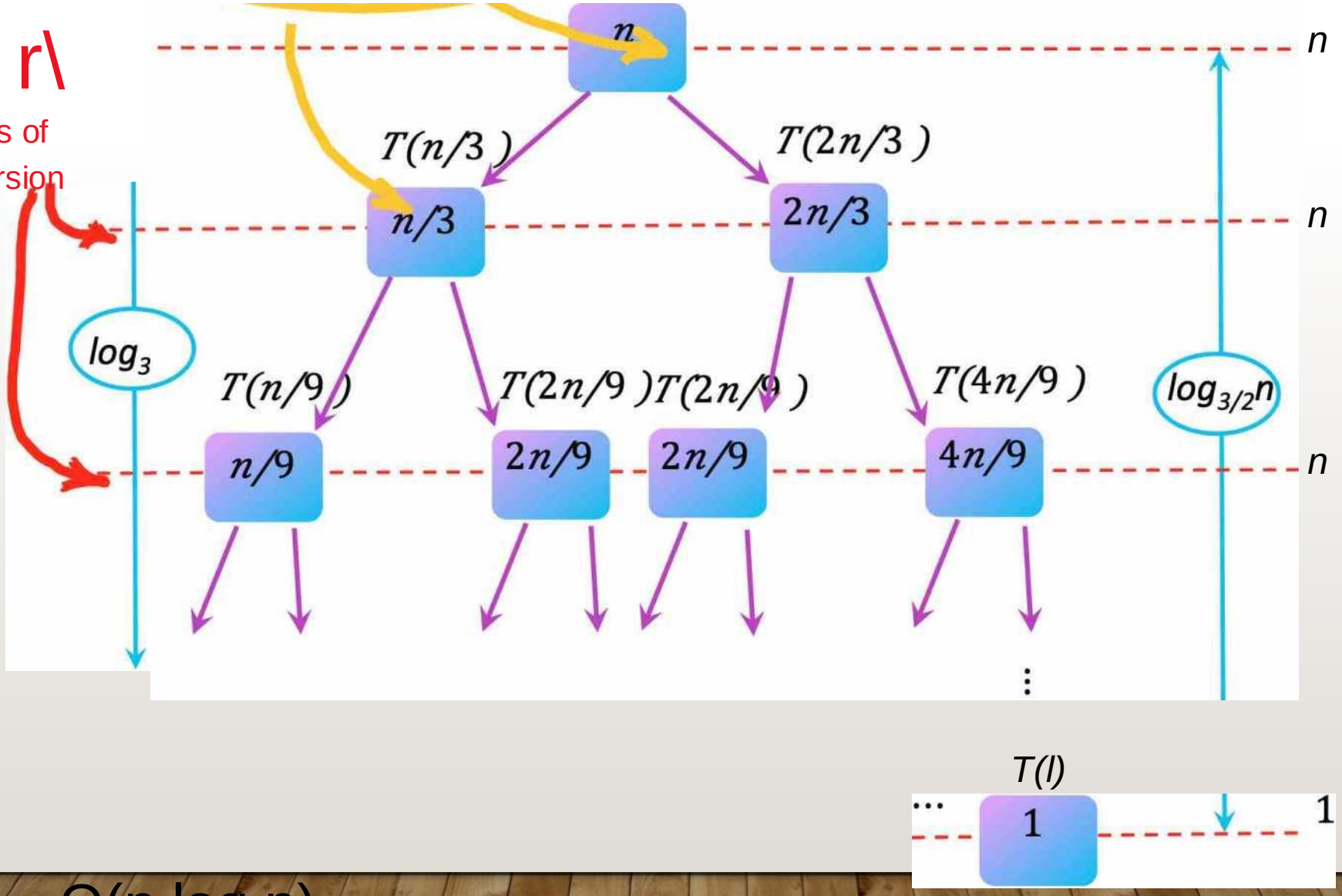
$T(n) = T(n/3) + T(2n/3) + n$

Amount of work per level

Amount of work

r\

Levels of Recursion



O(n log n)

Master Theorem: slightly simplified

Simplified setting for many D&C algorithms

Let a=# of subproblems, *(n/b)* = size of the subproblems (assume ceiling function for *n/b)*

and n the total size if

if T(n) = $aT(n/b) + O(n^d)$, then

$$T(n) = \begin{cases} \text{A. } O(n^d) \text{ if } d > \log_b a \\ \text{B. } O(n^d \log n) \text{ if } d = \log_b a \\ \text{C. } O(n^{\log_b a}) \text{ if } d < \log_b a \end{cases}$$

Proof Sketch

assume w.l.o.g. that $n = b^j$, for some *j.*

The work carried out on level *k* of the recurrence tree is $a^k * 0(n/b^k)^d$, where

number of subproblems — their sizes — work due to $0(n^d)$

$a^k * 0(n/b^k)^d = 0(n^d)(a/b^d)^k$ where $k=0,\ldots,\log_b n$ →

geometric series with ratio a/b^d

Proof sketch cont'd

Three cases arise:

(1) $a/b^d < 1$ -> series is determined by the first term

i.e., $O(n^d)$ *[work is decreasing as we go down the levels]*

(2) $a/b^d = 1$ -> all $O(\log n)$ terms are equal to $O(n^d)$

(3) $a/b^d > 1$ series (work per level) is increasing and its sum is determined by the last term $O(n^{\log_b a})$

3. cont'd

$$O(n^d (a/b^d)^{\log_b n}) = O(n^d(a^{\log_b n} / (b^{\log_b n})^d))$$

$$= O(a^{\log_b n})$$

$$= O(a^{\log_a n \log_b a})$$

$$= O(n^{\log_b a})$$

Example of uses of Master Theoreml

Binary Search

$$T(n) = T(n/2) + 0(1)$$

[again take n/2 to mean ceiling(n/2)]

What are a, *b,* and *d* for this example?

Binary search use of Master Theorem 1

a =1 $b = 2$, $d = 0$

now calculate $log_b a = \log_2 1 = 0$

We are therefore in Case B and thus get:

T(n) = $0(n^d log n)$ = 0(log n) as d=0

Binary search is therefore 0(log *n).*

Revisit MergeSort

- T(n) = $aT(n/b) + O(n^d)$, $a = 2$, b=2 , d = 1

$log_b a = log_2 2 = 1$ which equals d

A. $O(n^d)$ if $d > log_b a$

B. $O(n^d \log n)$ if $d = log_b a$ <- O(n log n) since d=1

C. $O(n^{\log_b a})$ if $d < log_b a$

Master Theorem I gives us rapidly O(n log n)

Solving Recurrences
Master theorem 2 - slightly more general

Let $a \geq 1$ and $b > 1$, left $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = a*T(n/b) + f(n)$

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

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

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

Note

this setting is more precise and also incorporates Θ bounds.

Working with the Master Theorem takes practice!

Example: $T(n) = 9T(n/3) + n$

What do you expect?

Apply Master Theorem with

$a=9$, $b=3$ and $f(n) = n$

$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$

Since $f(n) = O(n^{\log_3 9-\varepsilon})$ with $\varepsilon = 1$, we are in Case 1. Therefore: $T(n) = \Theta(n^2)$

Example: $T(n) = T(2n/3) + 1$

What do you expect?

Apply Master Theorem with a=1, b=3/2

and f(n) = 1

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

Since $f(n) = O(n^{\log_b a}) = \Theta(1)$, we are in

Case 2. therefore: $T(n) = \Theta(\log n)$

Example: $T(n) = 3\,T(n/4) + n\log n$

- What do you expect?
- Apply Master Theorem

  with a=3, b=4 and f(n) = n log n

  $n^{\log_b a} = n^{\log_4 3} = \Theta(n^{0.793})$

  Since $f(n) = \Omega(n^{\log_4 3+\varepsilon})$ with ε=0.2, we are in not quite yet in Case 3.

We need to show that a*f(n/b) < c f(n) for some constant

c <1 and all sufficiently large n.

Then, we will get: $T(n) = \Theta(f(n) = \Theta(n\log n)$

cont'd: $T(n) = 3\,T(n/4) + n\log n$

We need to show that af(n/b) < c f(n), for some constant c <1 and all sufficiently large n.

a*f(n/b) = 3*f(n/4) =

3*(n/4)log(n/4) ≤ (3/4)nlogn = (3/4) f(n), c=3/4

Therefore, we now get: T(n) = Θ(f(n) = Θ(nlog n)

Limitations of Master Theorem

Example: $T(n) = 2\,T(n/2) + n \log n$

- What do you expect?
- We cannot apply the Master Theorem.
- Attempt:

  with a=2, b=2 and f(n) = n logn

  $n^{\log_b a} = n^{\log_2 2} = \Theta(n)$

Case 3 does NOT apply.

"If $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for constant $\varepsilon > 0$, "

$f(n) = n \log n$ and $n^{\log_b a+\varepsilon} = n^{1+\varepsilon}$.

Limitations of Master Theorem cont'd

we have $f(n) = n \log n$ and $n^{\log_b a+\varepsilon} = n^{1+\varepsilon}$

Is $n \log n = \Omega(n^{1+\varepsilon})$ or not?

Note: $n \log n / n = \log n$

$\Omega(n^{1+\varepsilon})/n = \Omega(n*n^{\varepsilon}/n) = \Omega(n^{\varepsilon})$

but $\log n$ is not $\Omega(n^{\varepsilon})$.

[$\Omega(n^{\varepsilon})$is exponential and log n only logarithmic.] So, the Master Theorem cannot be applied here – GAP!

Limitations of Master Theorem very simplified view

1. If $f(n) = O(n^{\log_b a-\varepsilon})$ for constant $\varepsilon>0$, …

   f(n) “is less than” $n^{\log_b a-\varepsilon}$

2. If $f(n) = \Theta(n^{\log_b a})$ ….

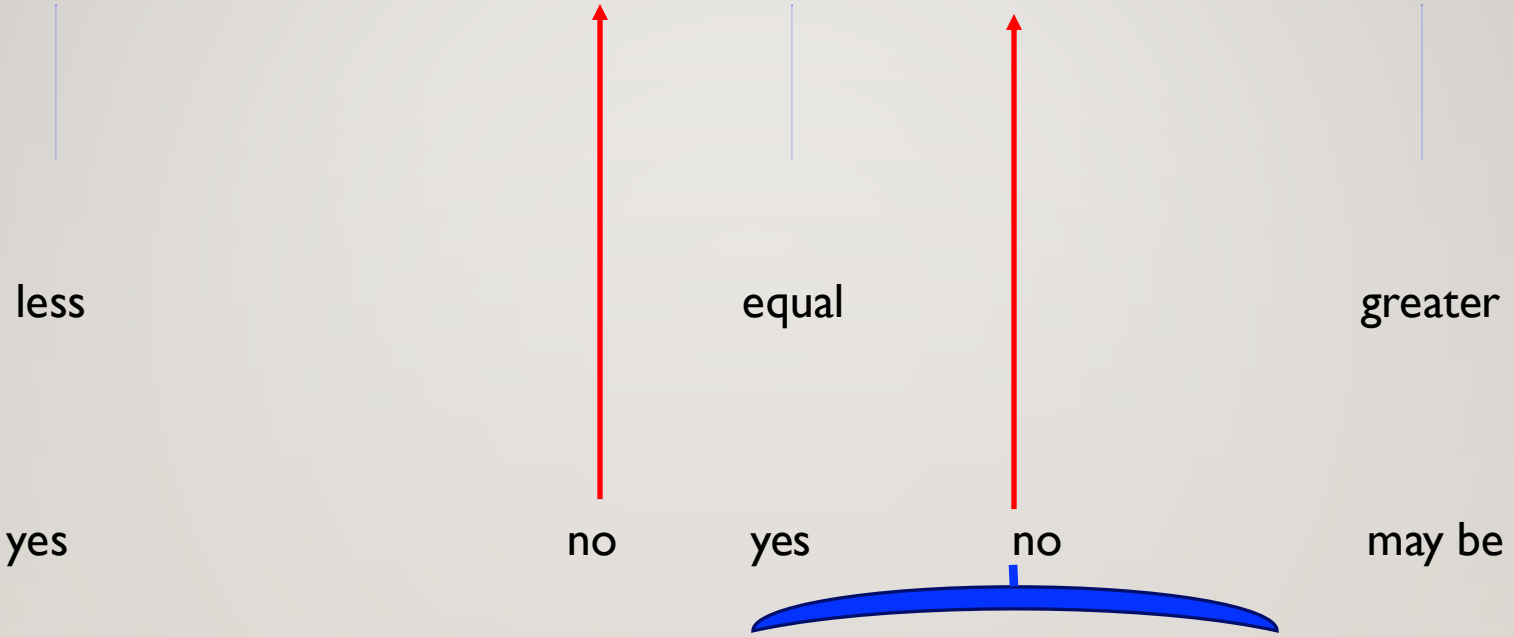
   f(n) “is equal” to $n^{\log_b a}$

3. If $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for constant $\varepsilon>0$, …

   f(n) “is greater than” $n^{\log_b a+\varepsilon}$

Where does the Master Theorem help us?

f(n): …. $(n^{\log_b a-\varepsilon})$ … $n^{\log_b a}$ …. $n^{\log_b a+\varepsilon}$ ….



There are 2 "no"s of applicability. We saw a=2, b=2 and f(n) = n logn

Exercise: construct an example for the other "no" gap of applicability..

Order Statistics

Objective:

— find the 1^{st}, 2^{nd}, median, or k^{th}, or n^{th} element in a set of n elements

— We know how to find the 1^{st} and n^{th} element in linear time, O(n). This is finding the minimum and finding the maximum of a set.

Order Statistics

The median is informally defined as the number in the set for which half of the numbers are smaller and half are larger.

More precisely,

— The median (lower median) of a set of n numbers is the element in position $\lfloor n/2 \rfloor$ after sorting the elements in increasing order.

— Motivation - see class

Median cont'd

The median (upper median) of a set of n numbers is analogously defined (replacing the floor function by the ceiling function).

Let us design several algorithms for this important order statistics.

I. Brute Force - see class
II. A randomized D&C algorithm
III. A worst-case optimal algorithm

Order statistics cont'd

- In fact, we will do more, we show how to compute the k^{th} order statistics, so the median is just a special case.
- We call the algorithm Selection(S,k)
- Input: a set S of n elements [orderable]

  and a parameter k with $1 \leq k \leq n$.

  Output: the k^{th} smallest element of S

Selection cont'd

$$\text{Selection}(S,k)= \begin{cases} \text{Selection}(S_L, k) \text{ if } k \le |S_L| \\ v \text{ if } |S_L| < k \le |S_L| + |S_V| \\ \text{Selection}(S_R,\ k - |S_L| - |S_V|) \text{ if } k > |S_L| + |S_V| \end{cases}$$

cont'd

where:

- $S_L^{=}$ {x in S | x < v}
- $S_v^{=}$ {x in S | x = v}
- $S_R^{=}$ {x in S | x> v}

Example: see class

1. Correctness

The algorithm is correct

Induction on |S|

| S | ={v} = 1 then k=l and the element v is returned

Now assume that the algorithm correctly computes the k^{th} order statistic for any set S where ISI <n.

proof of correctness Select

- Now consider a set S of cardinality n.
- First, the algorithm must determine the sizes of the sets S_L and S_R The correctness of this would have to be determined separately.
- If $k \leq |S_L|$ then the k^{th} order statistic lies in S_L.
- The algorithm recurses correctly on S_L.
- Otherwise, if $|S_L| < k \leq |S_L| + |S_v|$ then the k^{th} order statistic is one of the repetitions of the element v. This is correctly returned.

proof of correctness Select cont'd

if $k > |S_L| + |S_v|$ then the k^{th} order statistic lies in S_R. Select is thus called on S_R. But we need to subtract the size of S_L and the number of times v occurs for the call. The reason is that these elements are all smaller than the smallest element in S_R. This is done correctly by the algorithm.

2. Time Complexity

- The following recurrence describes the time complexity of Selection(S,k)

If we were lucky and v were the median

then the k^{th} order statistic could be computed in

$$T(n) = T(n/2) + 0(n)$$

using the simplified Master theorem

A. $O(n^d)$ if $d > log_b a$ applies

because a=1, b= 2 and d=1; $1 > log_2 1 = 0$

Thus T(n) = n.

Time Complexity cont'd

what is we are unlucky and the sets are very unbalanced.

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

then $T(n) = n^2$.

3. Can we improve upon this?

- Yes, in two ways

1) we show that a randomized algorithm (similar to Quicksort) computes the k^{th} order statistic in *expected* O(n) time.

2) a more complex, but deterministic algorithm exists that has O(n) runtime.

analysis to lead to improvement

Runtime of above algorithm depends on the sizes of the sets, in particular on S_L, S_R

Let us assume, for ease of notation, that each element is unique, so $S_v$has one element.

CASES arise: depending on the relative sizes of S_L, S_R

cont'd

1. if we are lucky: $|S_L| = |S_R|$

   then $T(n) = T(n/2) + O(n) = ... = ??$ see class

2. if we are very unlucky, then one of S_L, S_R is empty. Then, e.g., for the median we get $T(n) = O(n) + O(n-1) + ... + O(n/2) = O(n^2)$ see class

cont'd

3. "We are sort of lucky"

   v is called *good* if it falls between the 1⁄4n and 3⁄4n element in sorted order.

   Then, T(n) < T(3⁄4n) +O(n) = O(n)

   This is still fine to get a fast algorithm! A constant fraction of the number of elements gets removed in each iteration. The analysis as to why this is O(n) is similar to that of why T(n) = T(n/2) + O(n) = O(n)

cont'd

Lemma: On average a fair coin needs to be tossed two times before "heads" is seen.

Proof: Let E = expected number of tosses before heads is seen then E = 1 + 1/2E => E = 2

1 toss half the cases

done we recurse

Thus, after 2 split operations, on average, the array will shrink to at most 3⁄4 of its size.

cont'd

- A **randomly!** chosen v has probability of 0.5 to be good. So, 2 random choices of v on average suffice.

Theorem: The expected run-time of the Selection Algorithm is O(n).

Deterministic Algorithm

- Now, we show that we can always achieve O(n) run-time for median or any k^{th} order statistic. (so not only randomized, but determinsitic)

Select(S,k)

1. Divide the n-element input array into n/5 groups of 5 element each + one group containing the remaining n mod 5 elements.

EXAMPLE see class

algorithm cont'd

2. Find the Median of each of the 5 element group by sorting them (say insertion sort O(1))

3. Recursively find the median, x, of the floor(n/5) medians from 2.

4. Partition the input array around x

   Let j:= 1+ number of elements ≤ x; call the Set S_L

   Let n-j:= remaining elements; call the set S_R

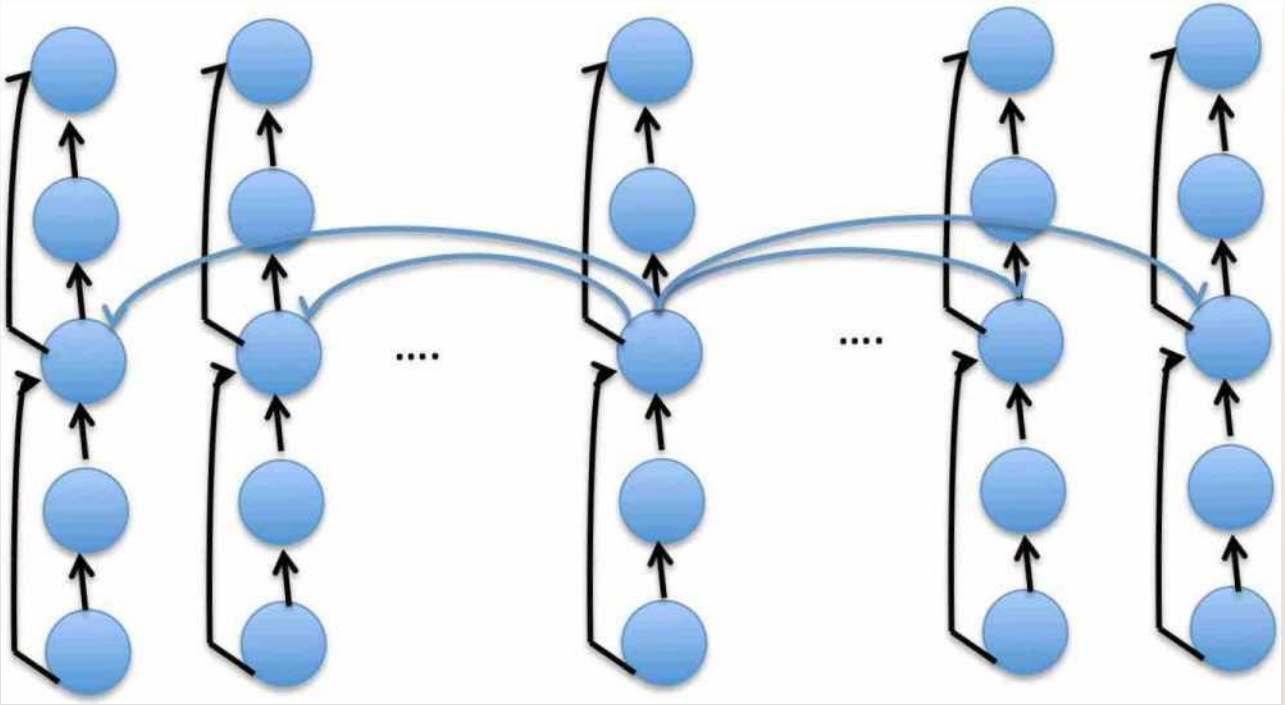
algorithm cont'd

5. if j = k then return x

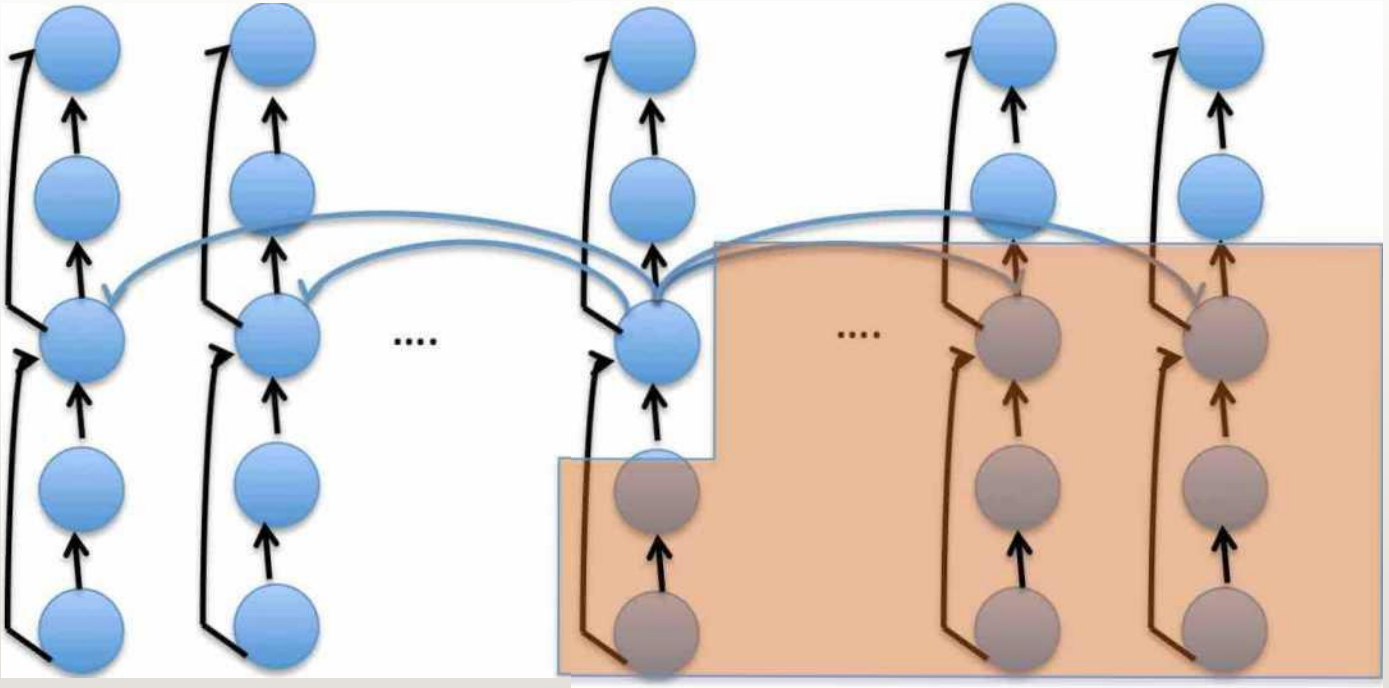
   if j < k recursively call Select(S_R , k-j)

   else j > k call Select(S_L , k)

Picture



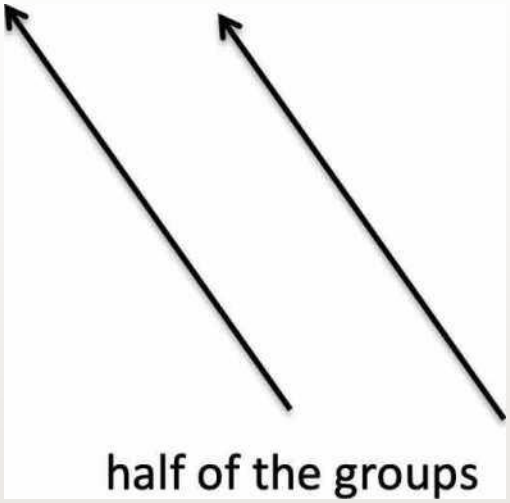
Picture



> x

analysis

- $3\left[\left\lceil 1/2\left\lceil n/5\right\rceil\right\rceil - 2\right] \geq 3n/10 - 6 \geq$ # of elements



guaranteed to be ≥ x

is thus 3n/10 - 6

two groups with fewer than 5 elements not counted

analysis cont'd

analogously for the # of elements guaranteed to be ≤ x.

Analysis cont'd Steps 1, 2

and 4 O(n)

Step 3 T([n/5])

Step 5 T(7n/10 + 6) why?

analysis cont'd

- 7n/10+ 6 < n for n>20
- Sorting say 80 numbers is 0(1)

$$* \; T(n) = \begin{cases} \Theta(1) \text{ if } n \leq 80 \\ T([n/5]) + T(7n/10+6)+O(n) \quad n > 80 \end{cases}$$

analysis cont'd

- $T(n) \leq c\lceil n/5 \rceil + c(7n/10+6) + O(n)$

  $< cn/5 + c + 7cn/10 + 6c + O(n)$

  $< 9cn/10 + 7c + O(n)$

  $< cn$ for some c

by def. of ”O"

T(n) = O(n)

Result

Theorem: Selecting the k^{th} element in a set of n elements takes O(n) time.

The k^{th} order statistic can be found in linear time (in the size of the set).

QUESTION: why does this method NOT work with groups of 3 elements (instead of 5)?

Extensions

In CS, we often do not only want to find particular values (such as Min, Max, Median, k^{th} order statistics), but maintain them dynamically under insertions and deletions.

See also dynamic binary search trees which are designed for efficient *searching* when we allow insertions and deletion.

Balanced Binary Search Trees

Downside

they take $\Theta(n \log n)$ time to build.

- $O(n \log n)$ is clear (or?)

- Why also $\Omega(n \log n)$?

  Think about our sorting lower bound here!

Maintaining order statistics

- We already know a data structure to maintain

  the min (or max) under a sequence of insertions and deletions.

Which one?

Heaps

The Priority Queue organizations such Heaps do that.

E.g. MinHeap on n elements:

- *FindMin* O(1)
- *DeleteMin* O(log n)
- *Insert*(a new element) O(log n)
- *CreateHeap* O(n)

analogously for MaxHeaps

Examples and review of operations

- see class
  - — we will review these operations that you should have seen before and analyze the performances in class.
  - - see textbook or your old notes

How fast can we find the maximum in a MinHeap or the minimum in a MaxHeap?

- *FindMin* O(1)
- *DeleteMin* O(log n)
- *Insert*(new) O(log n)
- *Create* O(n)
- *FindMax* ???
- *DeleteMax* ???

Where is the (a) maximum element?

Such an element must be at a leaf position in the MinHeap.

How many leaves does the tree have? approximately n/2

therefore *FindMax* here is Θ(n) !!

cont'd

analogously, the minimum element in MaxHeap is at one of the approx, n/2 leave positions -> *FindMin* in a MaxHeap takes O(n) time.

Why would we care about finding both Min and Max in a set?

Many reasons incl. the ones presented next!

How about maintenance of the median?

- Let us first show how to create a heap that allows the maintenance of the median under insertions (new) and deletions (of Median).

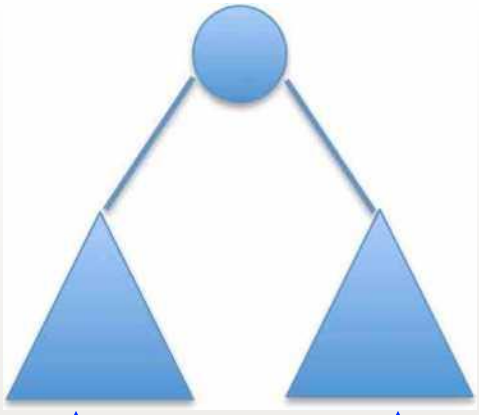
Let S be a set of n elements (stored in an array).

CreateMedianHeap(S) - first attempt

1. *FindMedian* using Select(S, floor(n/2))
2. Partition S around the median-> S_L , S_R
3. *Create*: *MaxHeap* on S_L and *MinHeap* on S_R

Median-heap

Median at root position



MaxHeap on S_L

MinHeap on S_R

Operation DeleteMedian

If we *delete* the Median

Where is the new median?

Depends on the relative sizes of S_L and S_R.

It is either the largest element in S_L or the smallest element in S_R.

Finding one of these is one of the efficient operations in the MaxHeap(S_L) and MinHeap(S_R).

Now how to insert?

- We insert by first comparing the element, called new, to the median.

  If new ≤ median then new is inserted in the left heap, i.e., the MaxHeap

  else new is inserted in the right heap, MinHeap

  After insertion, the Median may no longer be right.

cont'd

- we check the relative sizes of the left and right heaps.
- If the left heap is too heavy, we move the Median into the right heap and put the max of the left heap into the Median position.
- if the right heap is too heavy we move the Median into the left heap and replace it by the min of the right heap

Problems

A problem arises when we try to perform also the operations *DeleteMin*/*FindMin* and *DeleteMax*/*FindMax*

Where is the (global) minimum in the tree?

It must be in the left heap. Unfortunately, that is a MaxHeap which makes the operations *DeleteMin*/*FindMin* very inefficient.

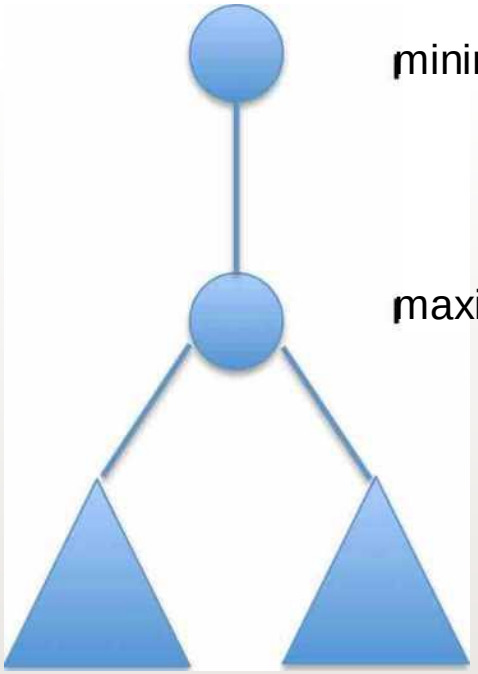
Analogously, for the (global) maximum.

How to solve this?

We need a heap like data structure that supports:

- *FindMin* O(1)
- *DeleteMin* O(log n)
- *Insert*(new) O(log n)
- *Create* O(n)
- *FindMax* O(1)
- *DeleteMax* O(log n)

MinMax-Heaps



minimum of all elements

maximum of all elements

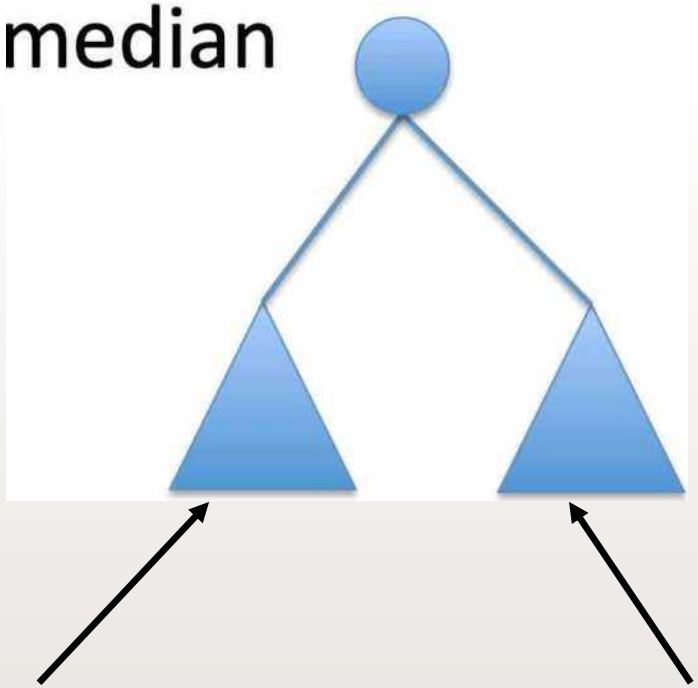
recursively alternating between min and max of the elements in the subtrees

Min-Max Heap

Example

— see class

Back to the Median heap



Min-Max heap
on elements ≤ median

Min-Max-heap
on elements > median

Creation

A comment on how to build/create the heap

1. *FindMedian*
2. Partition the elements around the Median call the sets (as always) S_L and S_R.

   (partitioning step looks familiar?)
3. Build two Min-Max Heaps on sets S_L and S_R.

How fast can we *Insert* into a Min-MaxHeap?

The operation, *Insert*(new), is very similar to that of a "normal" heap. That is bottom up. The difference is that we first determine if we are on a max- or a min-level. Then, we perform a possible swap with our parent node as required to keep the correct order. This is then followed by "bubbling up" two levels (instead of one as in normal heaps) towards the root as far as required.

How fast can we *Create* a Min-MaxHeap?

The operation is also similar to that of creating a "normal" heap. That is top-down. Again, the difference is that we first determine if we are on a max- or a min-level. Then, we perform a possible swap with our child node as required to keep the correct order. This is then followed by "trickle-down" two levels (instead of one as in normal heaps) towards the leaves as far as required.

Example

- in class

Median-Min-Max-Heap

Now, we have a heap that allows all these operations to be performed efficiently

- *FindMin* O(1)
- *FindMax* O(1)
- *FindMedian* O(1)
- *DeleteMin* O(log n)
- *DeleteMax* O(log n)
- *DeleteMedian* O(log n)
- *Insert*(new) O(log n)
- *Create* O(n)

Other operations on Priority Queues

(Recall) Priority queue organization

are used for maintaining job queues

in OS under insertions of new jobs and jobs get removed to be executed

the key is some prioritizations of the jobs

Merging

A useful operation is *Merge*

- *Merge*(P1, P2) // merges two priority queues (PQ) //
- Let n, k k≤ n be the sizes of the two PQs
- Let k-heap be the heap on the k elements
- and n-heap be the heap on the n elements
- Output: (n+k)-heap containing all elements.

Let us assume the two priority queues are implemented using heaps.
Then, how fast could we merge two heaps?

Algorithms to Merge

Algorithm 1

— for each element of k-heap do

*Insert(*element) into n-heap to create the (n+k)-heap

Complexity:

each insertion into the (n+k)-heap takes O(log(n+k))

since k≤ n this is O(log n).

Total: k*O(log n)

Is this the best we can do?

Algorithm 2

Assume in Algorithm 1 that k is O(n) then Algorithm 1 runs in O(n log n) time.

But, we can *Create* a heap on all elements in O(n+k)= O(n) time .

So, this is odd! By ignoring all structure and creating the heap from scratch we get a better algorithm (sometimes).

Comparison

So, when is Algorithm 1 better than Algorithm 2?

when is $k \log n \leq n$?

this means when $k \leq n/\log n$.

Can we do better?

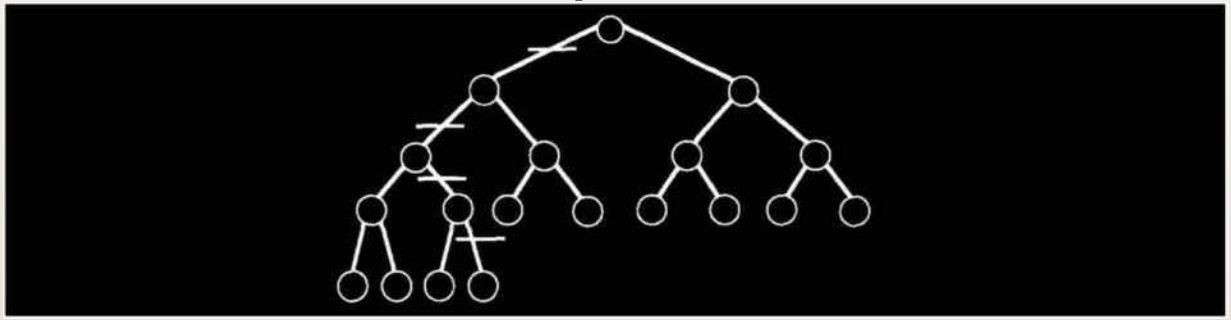
Let us count only comparison, so no data movements.

Brief Overview

no proofs will be required from you, but you need to understand the ideas

See the paper on the course web-site for more information.

Assume we have a heap like this n=19:



Indicated is the path to the last element. We cut each link on that path.

Merging Heaps

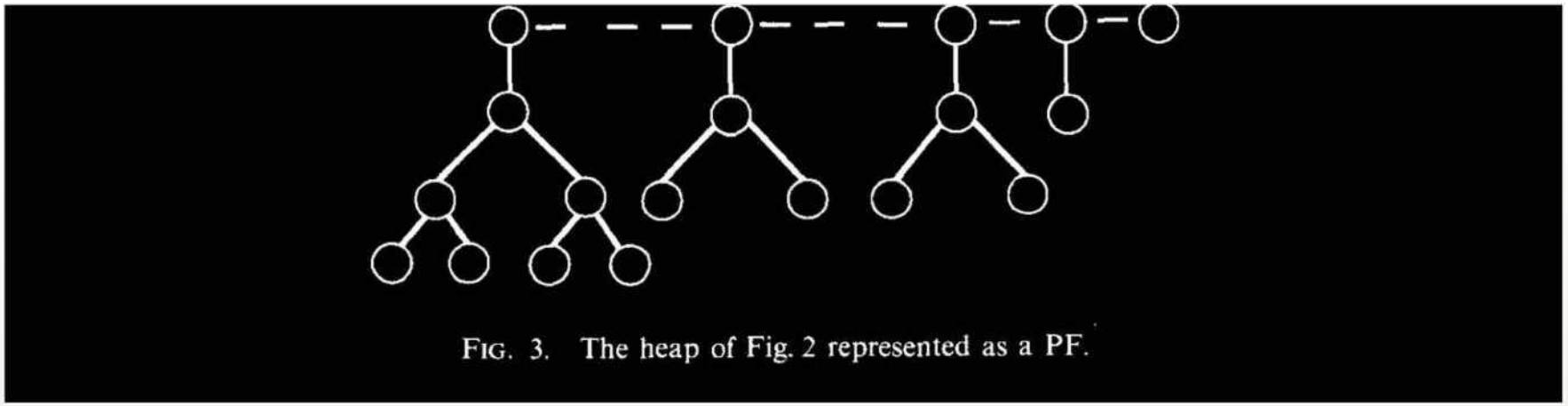


FIG. 3. The heap of Fig. 2 represented as a PF.

We organized the subtrees from the previous picture slightly differently. Note that

1) the roots are sorted
2) the subtrees have sizes 2^i for some i. call them *pennants*
3) there could be 0,1 or 2 “pennants” of the same size

Observations cont'd

If n is the size of heap then its decomposition into a forest of pennants is unique.

We can count the number of pennants of size 2^i and call that number d_i. As said above, $0 < d_i < 2$

There are at most $m = \log_2 n$ pennants in total.

Let $D = (d_m, ..., d_0)$ be the vector describing the unique decomposition.

For the above example, we get 19=(1,2,1,1)

cont'd

In some sense this is a new number representation where each digit is 0,1, or 2.

Now for two input heaps with n and k elements we can compute the descriptor.

Also, without even knowing the elements inside we know the structure of the merged (n+k)-heap because we can compute the descriptor.

Heaps

- Heaps are determined
  1) their shape
  2) their order

  Let us not worry about the order for now.

  1) shape:

Observation

Two pennants of the same size 2^i can be merged into a pennant of size 2^{i+1} in time proportionally to their heights.

First since $2^i + 2^i = 2^{i+1}$ the size of the resulting tree is 2^{i+1} and it is this a pennant.

How can they be merged in time proportionally to their height?

Exercise! (simple)

Construction idea

To construct the (n+k)-heap we need the pennants from the n-heap and the k-heap.

For this, we scan the descriptors of the three heaps in parallel from right to left.

The d_is of the n-heap and k-heap tell us which pieces we have the d_is of the (n+k)-heap tell us what we need.

Observation (informal)

There is an observation (Lemma) that would intuitively say the following:

- if, at some point i, the descriptor of (n+k)-heap needs pennants of size 2^i they will be available from the n-heap or k-heap (or both)
- if we have too many pennants left after that, they come in pairs are can be merged into the next larger pennant size, (like a carry)

Example

see class

Heap order

To fix the heap order is more involved.

You are not required to know this.

Vague idea here: (details in paper)
Essentially it involves ordering roots from the two pennant forest. We make sure that all roots of the n-heap pennants are ≤ all roots of the k-heap pennants, (assuming min-heaps)

This is done via trickle down operations on the pennants of the k-heap. Total cost: logn * log k

Theorem - no proof

Theorem: Two heaps of size n, k can merged in O(log n * log k) key comparisons.