Lecture 2021-03-02

Tuesday, 2 March, 2021 10:44

Starting today: Algorithms

- previously: TMs capture "universal computation"
- next few weeks: do things on a universal computer
  - but which ones? and what are we doing on them?

- Formally, an algorithmic problem is the task of computing some function $f: \Sigma^* \to \Sigma^*$ (restricted output to $\{0,1\}$)

- input $w \in \Sigma^*$ is an encoding of a "valid input"
  output $x \in \Sigma^*$ is also an encoding.

an algorithm is... some kind of "program" A s.t.

$$A(w) = f(w) \quad \forall w \in \Sigma^*.$$

What (Turing-complete) model will we assume?

- Unit-Cost RAM model
  - basic data type is an integer
  - numbers fit into "words"
  - arithmetic / comparison on words take constant time
    - bitwise ops / floors, ceilings require some care.
  - arrays allow random access
  - pointers store addresses in words

Caveats:

- sometimes we have situations where unit-cost makes no sense
  e.g. analyze arithmetic in terms of #bits
- assumptions only valid if alg's do not produce overly large intermediary values.
  large enough numbers need to broken up into multiple words

words

When all else fails, fall back to TMs.

Reductions $A \le B$

Informally, given an instance of problem A
convert it into an instance of problem B
apply known algorithm to problem B
convert output into correct solution for problem A

This is a <u>very powerful</u> algorithms design technique.
Instead of reinventing a tool, use someone else's work.

Quite often in this class the easiest way to come up w/ an algorithm is to reduce your problem to another problem w/ existing algorithm.

Example: given an array of integers, are there any duplicates?

Naive algorithm: double for loop:

```
for i from 1 to n
    for j from 1 to n:
        if A[i] = A[j]
            return true
```

$O(n^2)$

Better idea: reduce to sorting.

```
sort A
for i from 1 to n-1
    if A[i] = A[i+1]
        return true
```

Sort time $O(n \log n)$
$+$
$O(n)$

$O(n \log n)$

Next 2 ½ weeks: special kind of reduction called Recursion

Next 2 1/2 weeks: special kind of reduction called Recursion

we did a lot of recursion during automata!
hopefully you know by now:
recursion = induction

Recursion as an algorithmic technique:
reduce the problem of solving some instance of this problem to the problem of solving a smaller instance of the same problem
(self-reduction)

Tower of Hanoi

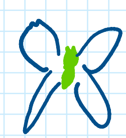
allowed to:
move one disk at a time
(only the top disk)
put smaller disks on top of larger disks.
goal: start w/ a stack of disks on peg & move it to another peg

disks, starting peg, ending peg, spare peg

Hanoi(n, src, dst, tmp)
  if n>0
    Hanoi(n-1, src, tmp, dst)
    move disk n from src to dst
    Hanoi(n-1, tmp, dst, src)

implicit: if n=0 do nothing.

recursion fairy



Running time of this alg? (count # moves)

T(n) = # of moves to get n disks from src to dst.

$T(n) = \#$ of moves to get n disks from src to dst.

$$T(n) = \begin{cases} 0 & \text{if } n=0 \\ T(n-1)+1+T(n-1) & \text{if } n>0. \end{cases}$$

Guess $T(n) = 2^n - 1$.

Base case: $n=0$, $2^0-1=0$. Assume for $k<n$ that $T(k)=2^k-1$. $T(n) = 2T(n-1)+1$ def $= 2(2^{n-1}-1)+1$ IH $= 2^n - 1$ math.	Assume for $k<n$ that $T(k)=2^k-1$ Two cases: • $n=0$: $T(0)=0=2^0-1$ • $n>0$: (same as left side).

Conclusion: $T(n) = O(2^n)$

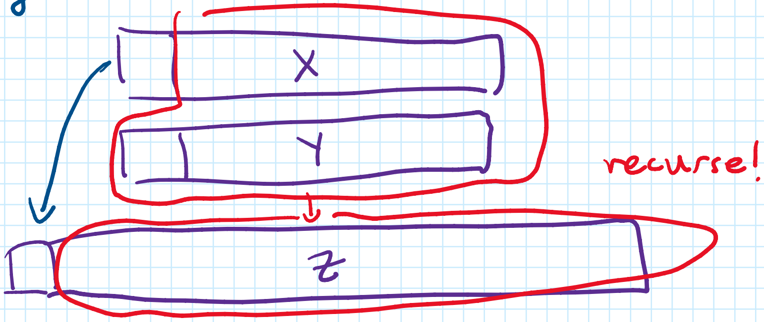
Merge Sort

Input	S	O	R	T	I	N	G	E	X	A	M	P	L
Divide	S	O	R	T	I	N	G	E	X	A	M	P	L
Recurse left	I	N	O	R	S	T	G	E	X	A	M	P	L
Recurse Right	I	N	O	R	S	T	A	E	G	L	M	P	X
Merge	A	E	G	I	L	M	N	O	P	R	S	T	X

MergeSort$(A[1..n])$
  if $n > 1$
    $m \leftarrow \lfloor n/2 \rfloor$
    MergeSort$(A[1..m])$ recursion fairy!
    MergeSort$(A[m+1..n])$ recursion fairy!
    Merge$(A[1..n], m)$ ← reduced to the problem of merging two sorted arrays.

Merge(A[1..n], m) ← reduced to the problem of merging two sorted arrays.

Merge:



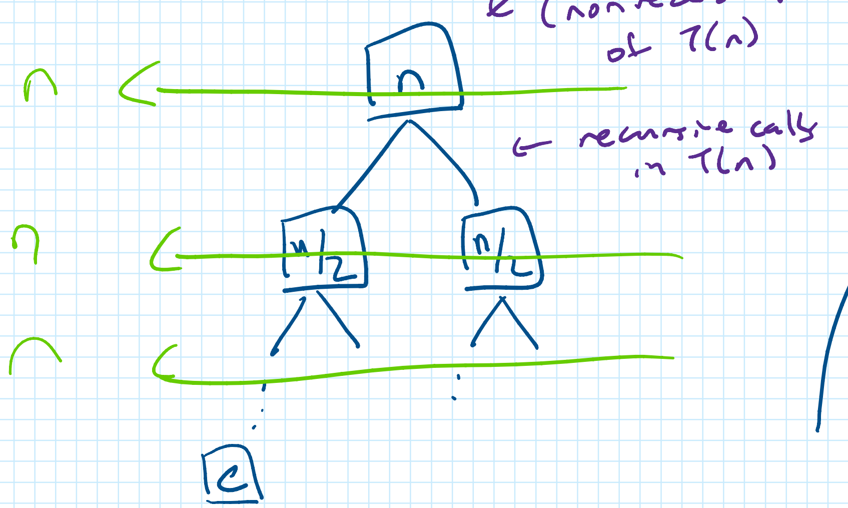
if X empty: set Z to be Y.

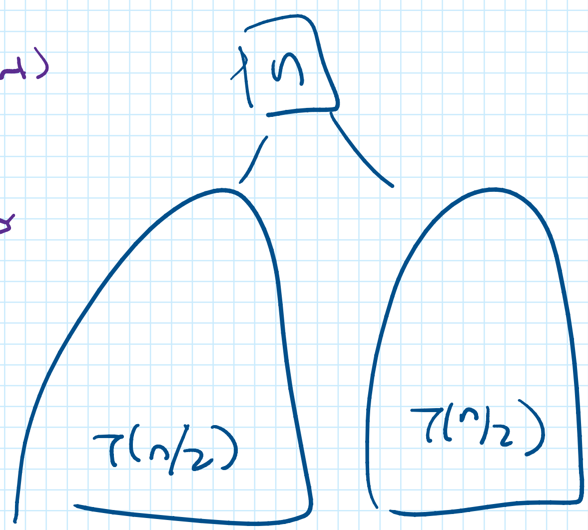
runtime of merge (exercise).

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$$

(justification for ignoring ⌈ ⌉ / ⌊ ⌋ for Big O → see Jeff's book)

Recursion Tree:





$$\left(\sum_{l=0}^{H-1} \underbrace{(\text{total work at level } l)}_{n} \right) + \text{base case work}$$

level l corresponds to $T(n/2^l)$

at level H, $T(n/2^H)$ is a base case

$n/2^H = \text{const.}$ solve for $H \to H = O(\log n)$

at level H: $T(n/2^H)$ is a base case

$n/2^H = \text{const.}$ solve for $H \to H = O(\log n)$

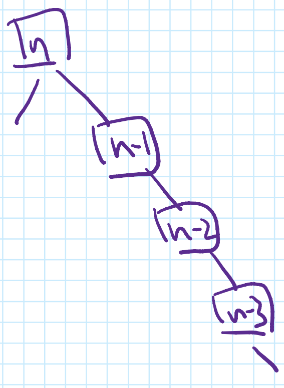
$$\sum_{l=0}^{O(\log n)} n \qquad 2^{O(\log n)} \text{ leaves.}$$

$$\boxed{T(n) = O(n\log n) + O(n) = O(n\log n)}$$

Worst case analysis of Quicksort

$$T(n) = O(n) + \max_r \left(T(r-1) + T(n-r)\right)$$

$$\le O(n) + T(0) + T(n-1)$$



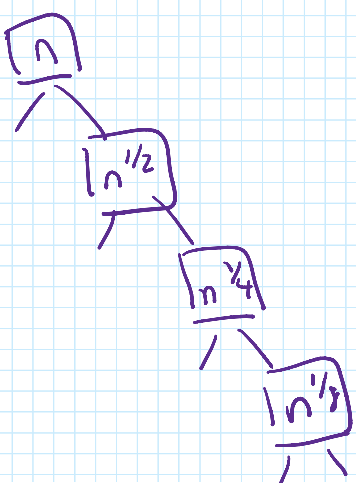
at level l: $T(n-l)$.

at level H: $n-H = \text{const}$

so $H = O(n)$.

$$\boxed{T(n) = O(n^2)}$$

$$T(n) = T(\sqrt{n}) + 1$$



at level l: $T(n^{1/2^l})$

H: $n^{1/2^H} = \text{const.}$

$\log(n^{1/2^H}) = \log(\text{const})$

$\to 2^{-H} \log n = \text{const}$

$\rightarrow 2^{-H} \log n = \text{konst}$

$\rightarrow \log 2^{-H} + \log\log n = \text{konst}$

$\rightarrow -H + \log\log n = \text{konst}$

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

const

$T(n) = aT(n-c) + \dots$

$\hookrightarrow O(n)$ levels

$T(n) = aT(n/c) + \dots$

$\hookrightarrow O(\log n)$ levels

$T(n) = aT(n^{1/c}) + \dots$

$\hookrightarrow O(\log\log n)$ levels