Final Test [December, 11, 2024; 4:00pm – 05:50pm]

1. Test 3 will include anything that we covered throughout the semester.
2. You can use practice test 1 and 2 as a ~~basis~~ basis of questions that will be covered in the test.
3. I will post a practice test 3 that includes questions from Big O and algorithm analysis as well (During the weekend)
4. You can bring any notes that you would like.
5. No Electronic devices.
6. Two hour test.
7. Expect more questions on the test.
8. Be prepared.
9. Today is the last day register for the test, If you wish to take it at the testing center.

Properties of Big O

lemma 6.2 Asymptotic equivalence of max and sum

we have.

$$f(n) = O(g(n) + h(n)) \iff f(n) = O(\max(h(n), g(n)))$$

Ex: $f(n) = n^2 + n = O(n^2 + n) \iff f(n) = O(\max(n^2, n)) = O(n^2)$

Lemma 6.3 Transitivity of $O(\cdot)$

If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then

$$f(n) = O(h(n))$$

ex: $f(n) = n + 5$, $g(n) = n$, $h(n) = n^2$

$f(n) = O(g(n)) = O(n)$, $g(n) = O(h(n)) = O(n^2)$

In other words,

$$[f(n) = O(g(n))] \wedge [g(n) = O(h(n))] \Rightarrow [f(n) = O(h(n))]$$

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

Lemma 6.4

Addition & multiplication preserve $O(\cdot)$-ness

$$[f(n) = O(h_1(n))] \wedge [g(n) = O(h_2(n))] \Rightarrow f(n) + g(n) = O(h_1(n) + h_2(n))$$

and

$$f(n) \cdot g(n) = O(h_1(n) \cdot h_2(n))$$

ex: $f(n) = n^2 + 1$, $h_1(n) = n^3$

$g(n) = n^4 + 2$, $h_2(n) = n^4$

$f(n) = O(n^3)$, $g(n) = O(n^4)$

$f(n) + g(n) = n^3 + n^2 + 1 = O(n^3 + n^4) = O(n^4)$

$f(n) \cdot g(n) = (n^3 + 1)(n^4 + 2) = n^6 + 2n^2 + n^4 + 2 = O(n^3 \cdot n^4) = O(n^7)$

Lemm 6.5

Let $f(n) = \sum_{i=0}^{k} a_i n^i = a_0 n^0 + a_1 n^1 + a_2 n^2 + \cdots + a_k n^k$

be a deg-k polynomial, then

$$f(n) = O(n^k)$$

$$f(n) = 100 \cdot n^6 + 50 \cdot n^3 + n^2 - 2n + 5$$

$$f(n) = O(n^6)$$

To measure runtime of an algorithm

1. find $f(n)$ by counting the # of steps
2. find the simplest $g(n)$ s.t $f(n) = \Theta(g(n))$

— Worst case running time analysis.

Example

```
sum = 0
for i = 1 to n do (1 + inner ops)
    for i = 1 to n do (1 + inner ops)
        for k = 1 to n (1 + inner ops)
            sum = sum + 1   <- 3 ops
return sum
```

$$f(n) = n \cdot (1 + n(1 + n(1 + 3)))$$

$$= n + n^2(1 + n + 3)$$

$$= n + 4n^2 + n^3$$

Ex 2

while $n > 1$ do $(n-1)$ (2 + inner ops)

$n = n - 1$ ← 3 ops

return n.

$$f(n) = (n-1)(2+3)$$
$$= 5n - 1$$

ex!

while $n > 1$ do $(\log_2 n)$ times

$n = n/2$ ← 3 ops

$$f(n) = 3 \cdot \log n$$

What if the running time depends on specific input of size n.

```
linearSearch(A[1 ... n], x):
Input: an array A[1 ... n] and an element x
Output: is x in the (possibly unsorted) array A?
1 for i := 1 to n:
2     if A[i] = x then
3         return True
4 return False
```

(a) Linear Search.

$A = (2, 3, 7, 8)$ $x = 5$ ← worst possible input.

$A = (1, 6, 2, 5)$ $x = 6$

$A = (7, 6, 5, 8)$ $x = 7$ ← Best possible input

All of these arrays are of same length.
However # of steps that we execute are different.

Note!
Even when the input size n is the same, program may take different number of steps.
So, run-time depends not just on input size, but also what the input is.

we could be

1. Optimistic — best case
2. pessimistic — worst case → easy to define, guarantee for all inputs
3. Neither — Average case

↑ very appealing
- but we have to know the distribution of the inputs.
- we need make assumptions about the frequency of each input.
- This is harder to do.

worst case running time Analysis.

- Intuitively: For each n (size of the input), what is the input that makes the running time ~~the~~ worst?

Def

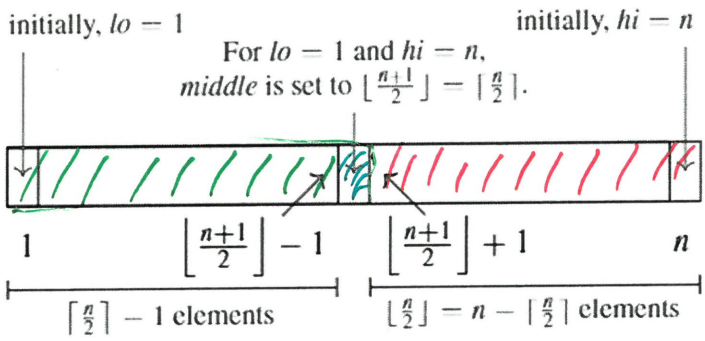
worst-case runtime of an algorithm is

$$T(n) = \max_{x:|x|=n} [\text{the number of primitive steps used by the algorithm on input } x]$$

Worst-case running time Analysis

```
binarySearch(A[1 ... n], x):
Input: a sorted array A[1 ... n]; an element x
Output: is x in the (sorted) array A?
1  lo := 1                                  ] c1 > 0
2  hi := n                                  ]
3  while lo ≤ hi:
4     middle := ⌊(lo+hi)/2⌋                 ]
5     if A[middle] = x then                 ]
6        return True                        ]
7     else if A[middle] > x then            ] c2 > 0
8        hi := middle − 1                   ]
9     else                                  ]
10       lo := middle + 1                   ]
11 return False                             ] c3 > 0
```

(b) The pseudocode for Binary Search.



(c) An illustration of the first split in binary search.

Figure 6.15 Linear and Binary Search.

worst case input for a array of size n

X not in A

best case input for a array of size n

X is in the middle of the array.

worst-case analysis

Claim - Worst case analysis of binary search is $O(\log n)$

Proof: The worst case running time occurs when the input x is not in the input array A. because while loop executes maximum # of iterations.

Note that at each iteration, while loop halves the # of elements under-consideration.

∴ # times that we execute the loop is $\log_2 n$

Before the while loop we execute c_1 ops

Then at each iteration we execute c_2 ops

After the loop we execute c_3 ops

overall, $f(n) = c_1 + c_2 \times \log_2 n + c_3$, $f(n) = O(\log n)$

Analysis of recursive algorithms.

Problem: factorial.
input : $n \in \mathbb{Z}^{>0}$
output : $n! = n \cdot (n-1) \cdot (n-2) \cdots \cdot 3 \cdot 2 \cdot 1$

Solution: ← name of the algorithm

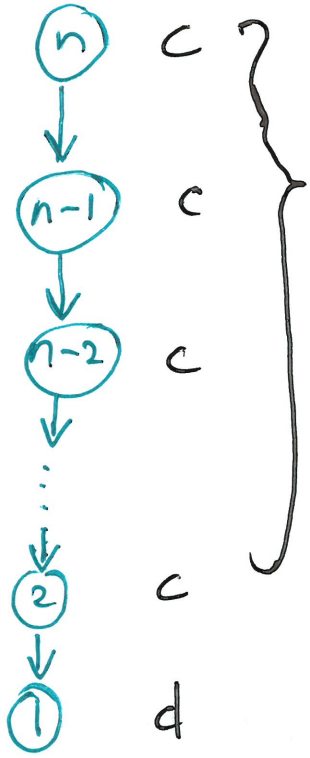
```
fact(n):
  if n==1 then
     return 1
  else
     return fact(n-1)·n
```

what is the run-time of the algorithm fact?

Idea 1: look at the recursion tree.

Def:
The recursion tree for a recursive Algorithm A is a tree that shows all of the recursive calls spawned by a call to A on an input size n. Each node in this tree is annotated with the amount of work, aside from any recursive calls, done by that call.

recursive tree for fact(n) on a input of size n.



runtime f(n) = total runtime of all nodes.

n-1 nodes with c ops.

```
fact(n):
d { if (n==1) then
      return 1
c { else
      return n.fact(n-1)
```

$$f(n) = c \cdot (n-1) + d$$
$$= cn + d - c$$
$$f(n) = O(n)$$

Idea 2! Using a recurrence relation.

∴ ...

Def: Recurrence relation

A recurrence relation (sometimes called a recurrence) is a function $T(n)$ that is defined (for some values of n) in terms of $T(k)$ for input values $k < n$.

```
fact(n)
  if n == 1          } d
    return 1
  else               } c
    return n · fact(n-1)
```

Ex! for fact(n), the runtime $T(n)$ is

$n > 1$ $T(n) = T(n-1) + c$, $n = 1$ $T(1) = d$

But, we need a closed form solution T(n).

How can we get a closed form solution from a recurrence relation.

non-recursive

① Iterate the solution a few times.

② Make a guess about closed form.

③ Try to prove the guessed claim.

How? Induction

$n > 1$ $T(n) = c + T(n-1)$, $n = 1$ $T(1) = d$

① $T(1) = d$ $= d$

$T(2) = c + T(1) = c + d$ $= c + d$

$T(3) = c + T(2) = c + (c + d)$ $= 2 \cdot c + d$

$T(4) = c + T(3) = c + (c + (c + d)) = 3 \cdot c + d$

② $T(n) = (n-1) \cdot c + d$

③ Use induction to prove $T(n) = (n-1)c + d$

Let us prove that running time of factorial algorithm is $T(n) = c(n-1)+d$, given that recurrence relation is:

$$T(n) = \begin{cases} d & n=1 \\ T(n-1)+c & n>1 \end{cases}$$

We use induction to prove this claim.

First, let us define predicate $P(n)$

$P: \mathbb{N} \longrightarrow \{T, F\}$, where $P(n)$ is defined as follows.

$$P(n) = \begin{cases} \text{True}, & \text{if } T(n) = c.(n-1)+d \\ \text{false} & \text{otherwise.} \end{cases}$$

we are trying to prove $[\forall n \geq 1 : P(n)]$

1. Base case is when $n=1$.
   using the recurrence relation $T(1) = d$
   using the formula $T(1) = c \times 1 + d - c = d$
   Therefore, the $T(n) = (n-1)\cdot c + d$ is true when $n=1$
   Base case is proven.

2. Inductive step

we want to show $[\forall n > 1 : P(n-1) \Rightarrow P(n)]$

Let $n > 1$

Assume $P(n-1)$ is true. ← Inductive hypothesis

This means $T(n-1) = c \times (n-2) + d$

consider $T(n)$,

using recurrence relation

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

$T(n) = c \cdot (n-2) + d + c$ (substituting inductive hypothesis)

$T(n) = c(n-1) + d$

↑

This means $P(n)$ is true.

$P(n)$

we have shown that $\forall n > 1 : P(n-1) \Rightarrow P(n)$

Therefore, using principle of mathematical induction, we can conclude that

$[\forall n \geq 1 : P(n)]$

Therefore, $T(n) = c \cdot (n-1) + d$ for factorial algorithm.