Lec 11: Recursion and Recurrence

Prof. Adam J. Aviv

GW

CSCI 1311 Discrete Structures I
Spring 2023

Algorithmic Performance

How do we compare two algorithms? Which one is faster?

```
int find(int x, int[] array){

  for(int i=0;i<array.length;i++){

    if (array[i] == x) return i;

  }

  return -1;
}
```

```
void sort(int[] array){

  for(int i=0;i<array.length;i++){

    for(int j=i+1;j<array.length;j++){

      if(array[j] < array[i]){

        int k = array[i]; //swap
        array[i] = array[j];
        array[j] = k;
      }
    }
  }
}
```

Counting Steps

Consider every operation as a "step." That is, any comparison, assignment, addition, etc. Then, how many steps does it take in the *worst case*?

```
int find(int x, int[] array){
  for(int i=0;i<array.length;i++){
    if (array[i] == x) return i;
  }
  return -1;
}
```

But it also depends on how long the array is. Let's assign an array length as the variable n.

Counting Steps: find()

```
int find(int x, int[] array){

  //1 step: int i = 0
  //1 step i< array.length
  for(int i=0;i<array.length;i++){
    //n iterations of ..

    //1 step: array[i] == x
    if (array[i] == x) return i;

    //1 step: i++
    //1 step i<array.length
  }

  return -1; //1 step return
}
```

$$S_{\text{find}}(n) = \overbrace{2}^{\text{int i=0;i<array.length}} + \underbrace{3 \cdot n}_{n \text{ iterations of: array[i]==x;i++;i<array.length}} + \overbrace{1}^{\text{return -1}}$$

$$S(\text{find}) = 3 \cdot n + 3$$

Counting Steps: sort

```
void sort(int[] array){
  //n steps for first loop

  //2 steps to initialize and compare
  //n iterations of ..
  for(int i=0;i<array.length;i++){\
    //2 steps to initialize and compare
    //n-1 iterations of
    for(int j=i+1;j<array.length;j++){
      //1 for comparison
      if(array[j] < array[i]){
        //3 for swap
        int k = array[i];
        array[i] = array[j];
        array[j] = k;
      }//2 steps to increment and
     compare
    }//2 steps to increment and compare
  }
}
```

$$S_{\textbf{sort}}(n) =$$

$$\overbrace{2}^{\textbf{int i=0;i<array.length}} + \overbrace{4\cdot n}^{n\times\textbf{int j=i+1;j<array.length}}$$

$$+ \underbrace{6\cdot(n-1))}_{i=\mathbf{0}:\ \textbf{6 steps}\ \times(n-\mathbf{1})} + \underbrace{6\cdot(n-2)}_{i=\mathbf{1}:\ \textbf{6 steps}\ \times(n-\mathbf{2})}$$

$$+ \quad \ldots$$

$$+ \underbrace{6\cdot 2}_{i=(n-\mathbf{2}):\ \textbf{6 steps}\ \times\mathbf{2}} + \underbrace{6\cdot 1}_{i=(n-\mathbf{1}):\ \textbf{6 steps}\ \times\mathbf{1}}$$

$$= 2 + 4\cdot n + 6\cdot\sum_{k=1}^{n-1} k$$

$$= 2 + 4\cdot n + \frac{6\cdot n(n-1)}{2}$$

$$= 2 + 4\cdot n + 3(n^2 - n)$$

$$= 3\cdot n^2 + n + 2$$

Comparing `find` and `sort`

Which routine is faster? That is, requires fewer steps in the worst case for an array of length n?

$$S_{\text{find}}(n) = 3 \cdot n + 3$$
$$S_{\text{sort}}(n) = 3 \cdot n^2 + n + 2$$

For big values of n (like really, really, big), n^2 will dominate n.

So `find` is faster than `sort`, requiring fewer steps in the worst case.

Big-O Notation

Definition

Big-O Let f and g be real value functions on the set of same negative real numbers, then we say f **is of order at most** g **written** $f(x)$ **is** $O(g(x))$, if, and only if, there exists a positive real numbers B and b such that:

$$(\forall x > b)\ f(x) < B \cdot g(x)$$

Another way to understand this definition is that for any function $f(x)$, we can identify a function $g(x)$ that is its upper bound.

For example, we can show that $f(x) = 3n + 3$ is in $O(g(x))$ where $g(x) = x$.

Converting to Big-O

Proof.

To prove $S_{\textbf{find}}(x) = f(x) = 3x + 3$ is in $O(g(x) = x)$, let $B = 10$ and $b = 19$. By induction on x, in the base case let $x = b + 1 = 20$ and $f(x) < B \cdot g(x)$

$$\begin{aligned} f(x) < B \cdot g(x)) &= 3 \cdot 20 + 3 < 10 \cdot 10 \\ &= 63 < 100 \end{aligned}$$

In the inductive case we need to show that

$$\begin{aligned} 3(x+1) + 3 &< 10(x+1) \\ 3x + 6 &< 10x + 10 \\ 3x - 4 &< 10x \\ 3x - 4 &< 3x + 3 < 10x && \text{by IH: } f(x) < B \cdot g(x) \equiv 3x + 3 < 10x \\ 3x - 4 &< 3x + 3 && \text{showing this, shows the result b/c } 3x + 3 < 10x \\ 3x - 3x &< 2 + 4 \\ 0 &< 6 \end{aligned}$$

Thus $S_{\textbf{find}}(x)$ is $O(g(x) = x)$, or more simply, $O(x)$. □

Exercises

Prove the following Big-O's:

$f(n) = 3n + 5$ is $O(n^2)$

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

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

A abbreviated understanding of Big-O

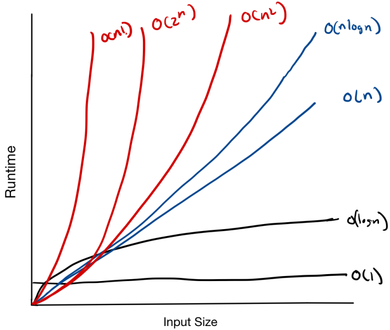
Once you do enough of these, you learn quickly that to prove something is in Big-O, you:

- Drop all constants – like 1 or 10 or 20
- Identify the dominate term – like n^2 or 2^n
- The Big-O is the dominate term – like $O(n)$ or $O(n^2)$

$$
\begin{aligned}
S_{\text{find}}(n) &= 3 \cdot n + 3 && \text{is } O(n) \\
S_{\text{sort}}(n) &= 3 \cdot n^2 + n + 4 && \text{is } O(n^2) \\
f(n) &= n^3 - n^2 + n - 300 && \text{is } O(n^3) \\
f(n) &= log(n+5) + 2 && \text{is } O(\log n) \\
f(n) &= 10n + 11\log(n) && \text{is } O(n) \\
f(n) &= 10n + n\log(n) && \text{is } O(n \log n) \\
f(n) &= 2^n + n^{100} && \text{is } O(2^n) \\
f(n) &= 42 && \text{is } O(1)
\end{aligned}
$$

Also, we want the smallest big-O that bounds a function.

Comparing Big-O's



$$\underbrace{O(1)}_{\text{constant}} < \overbrace{O(\log n)}^{\text{logarithmic}} < O(n \log n) < \underbrace{O(n^2) < O(n^3)}_{\text{polynomial}} < \overbrace{O(2^n)}^{\text{exponential}} < O(n!)$$

Big-O Logs

Under Big-O, we don't specify the log base because we can prove a log of any base is Big-O of a log of any other base. For example,

Proof: $f(x) = \log_{10}(x)$ is $O(\log_2(x))$.

Let $B = \dfrac{2}{\log_2(10)}$ and $b = 1$, then we need to show:

$$\log_{10}(x) < 2 \cdot \frac{\log_2(x)}{\log_2(10)} \quad \text{by Log Change of Base of Rule}$$

$$\log_{10}(x) < 2 \cdot \log_{10}(x)$$

$$1 < 2$$

□

And you can always choose a B of similar form for any change of base. Thus we simply just say $O(\log)$. And since we are CS people, we assume the log is base 2.

Exercises

What is the step counts and the Big-O of the following functions, assuming n as variable.

```
int sum = 0;
for (int i = 0; i < n; i++) {
  for (int j = 0; j < i/2; j++) {
    sum++;
  }
}
```

```
int sum = 0;
for (int i = 0; i < n/2; i++) {
  for (int j = 0; j < n/2; j++) {
    sum++;
  }
}
```

```
int sum = 0;
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n*n; j++) {
    sum++;
  }
}
```

```
int sum = 0;
for (int i = 0; i < n; i++) {
  for (int j = 0; j < i*n; j++) {
    sum++;
  }
}
```

Recursive Functions

What is the big-O of a recursive function? Assume the length `array` is n and it's called as `sum(0,array)`

```
int sum(int i, array[]){
  if(i >= array.length)
    return 0;
  else
    return array[i] + sum(i+1,array);
}
```

$O(n)$: Requires n recursive calls (the length of the array), and each call is a constant amount of work.

Recursion as recurrence

Consider that a recurrence relation is a lot like a recursive function. Let's use a recurrence to describe the step function for this routine.

```
int sum(int i, array[]){
  if(i >= array.length)//1 step
  return 0; //1 step
  else
  return array[i] + sum(i+1,array);
  //array[i] : 1 step
  //i+1 : 1 step
  //sum(i+1,array) : S_{n-1} (recurrence
     )
  // + : 1 step
  //return: 1 step
}
```

In the n-th recursion call, the steps S_n is

$$S_n = S_{n-1} + 5 \qquad \text{recursive case}$$

$$S_0 = 1 \qquad \text{base case}$$

Solving the recurrence for Big-O

$$
\begin{aligned}
S_n &= S_{n-1} + 5 && \text{recursive case} \\
S_0 &= 1 && \text{base case}
\end{aligned}
$$

Solving the recurrence:

$$
\begin{aligned}
S_n &= S_{n-1} + 5 \\
S_n &= S_{n-2} + 5 + 5 \\
\ldots \\
S_n &= S_{n-i} + 5i && i = n \text{ for base case} \\
S_n &= S_0 + 5n \\
S_n &= 5n + 1
\end{aligned}
$$

The Big-O of S_n is $O(n)$.

Recursion with loops

What is the step function, as a recurrence relation, that describes the following routine?

```
int sumsum(int i, array[]){
  if(i >= array.length){
    return 0;
  }else{
    int s=0;
    for(int j=0;int j<i;j++)
      s += array[j];
    return s + sumsum(i+1,array);
}
```

In the deepest, *n*-th, recursive call, there are *a* number of steps performed *n*-times, plus the amount in the recursion, plus some *b* more steps. Then *c* steps in base.

$$S_n = a \cdot n + S_{n-1} + b \qquad \text{recursive case}$$
$$S_0 = c \qquad \text{base case}$$

Determining Big-O

$$
\begin{aligned}
S_n &= a \cdot n + S_{n-1} + b \\
&= a \cdot n + a \cdot (n-1) + S_{n-2}) + b + b \\
&= a \cdot n + a \cdot (n-1) + a \cdot (n-2) + S_{n-3})b + b + b \\
&\cdots \\
&= a \sum_{j=0}^{i} (n-j) + S_{n-i} + i \cdot b && n = i \text{ in base} \\
&= a \sum_{j=0}^{n} (n-j) + S_0 n \cdot b \\
&= a \sum_{j=0}^{n} j + c + n \cdot b \\
&= a \cdot \frac{n(n+1)}{2} + c + n \cdot b \\
&= \frac{a}{2} n(n+1) + c + n \cdot b && \text{let } \frac{c}{2} = d \\
&= d \cdot n^2 + d \cdot n + d + c + n \cdot b \\
&= d \cdot n^2 + (d+b) \cdot n + d + c && \text{let } d + b = e; d + c = f \\
&= d \cdot n^2 + e \cdot n + f && \text{dropping constants} \\
&= n^2 + n && O(n^2)
\end{aligned}
$$

Exercises

Find the recurrence function, solve it, and then determine the Big-O for the routines below. Assume all functions are called as `foo(0,n)` for some n.

```
int foo(int i, int n){
  if(i > n){
    int k;
    for(k=0;k<n;k++);
    return k;
  }else{
    return 1 + bar(i+1,n);
}
```

```
int foo(int i, int n){
  if(i > n){
    return 1;
  }else{
    return 1 + bar(i+1,n) + bar(i+1,n);
}
```

```
int foo(int i, int n){
  if(i > n){
    return 1;
  }else
    return 1 + bar(i+1,n-1);
}
```

```
int foo(int i, int n){
  if(n==1){
    return 1;
  }else
    return 1 + bar(i+1,n/2);
}
```