Today’s class

- Analysis of algorithms
- Asymptotic notation
- Recurrence relation

Analysis of algorithms

Asymptotic notation

Big-Oh notation

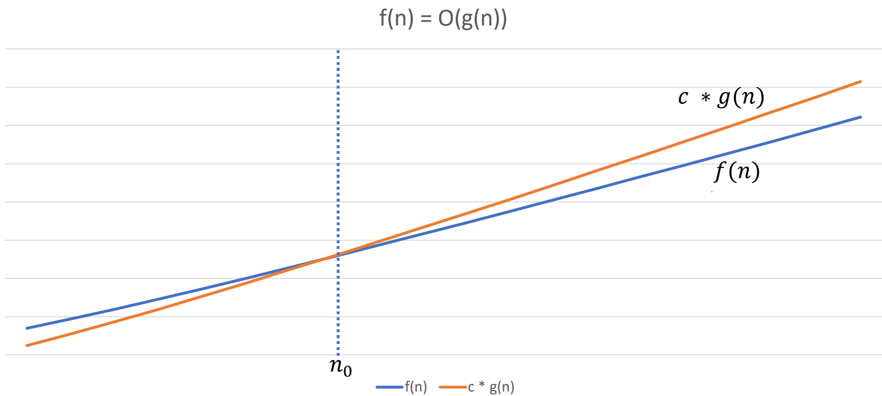
Let $f(n)$ and $g(n)$ are increasing functions of n

$f(n)\ is\ O\big(g(n)\big)$ (pronounced as order of)

if for some constants $c > 0$ and $n_0 \geq 1$

$$f(n) \leq cg(n), \qquad for\ all\ n > n_0$$

Big-Oh notation



c * g(n) is an asymptotic upper bound for f(n)

Big-Oh notation

- Big-Oh is used to describe asymptotic upper bound

- Even though there are many possibilities for the upper bound, we try to stay close to the real complexity
  - e.g., $3n^2 + 8 \ is \ O(n^4)$ but doesn't make much sense

- We remove the constant factors and lower order terms in the big-Oh notation

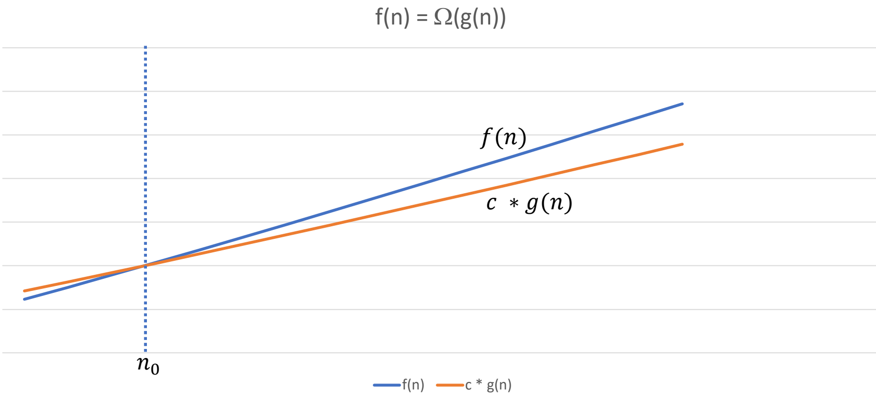
Big-Omega notation

Let $f(n)$ and $g(n)$ are increasing functions of n

$f(n)\ is\ \Omega\big(g(n)\big)$ (pronounced as big-Omega of)
if for some constants $c > 0$ and $n_0 \geq 1$

$$f(n) \geq cg(n), \qquad for\ all\ n > n_0$$

Big-Omega notation



c * g(n) is an asymptotic lower bound for f(n)

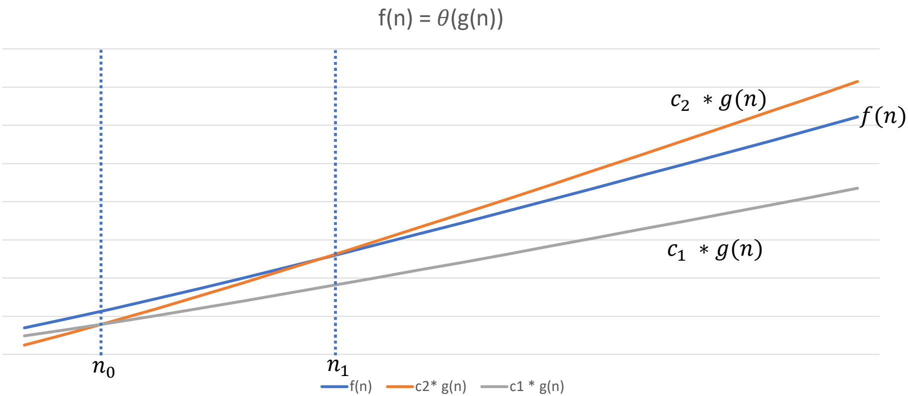
Big-Theta notation

Let $f(n)$ and $g(n)$ are increasing functions of n

$f(n)\ is\ \theta\big(g(n)\big)$ (pronounced as big-Theta of)

if $f(n)\ is\ both\ O\big(g(n)\big)$ and $\Omega(g(n))$

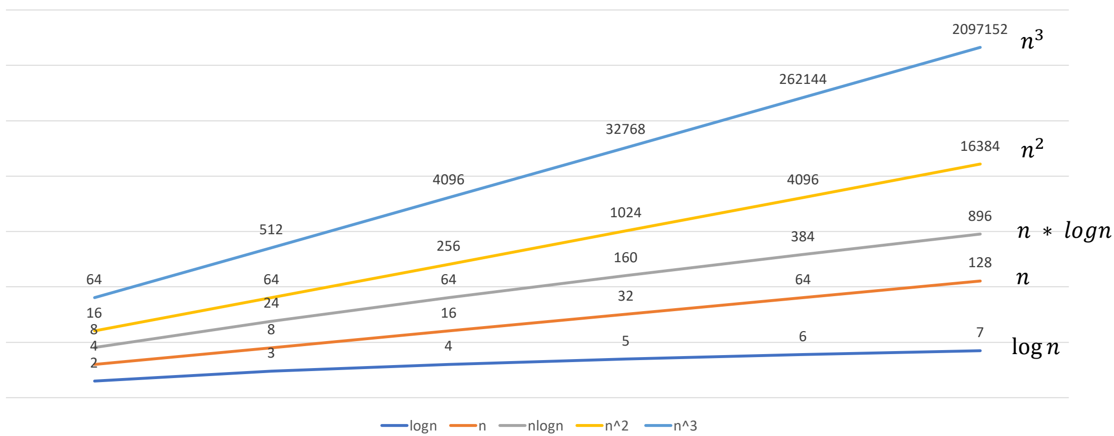
Big-Theta notation



Asymptotic notations

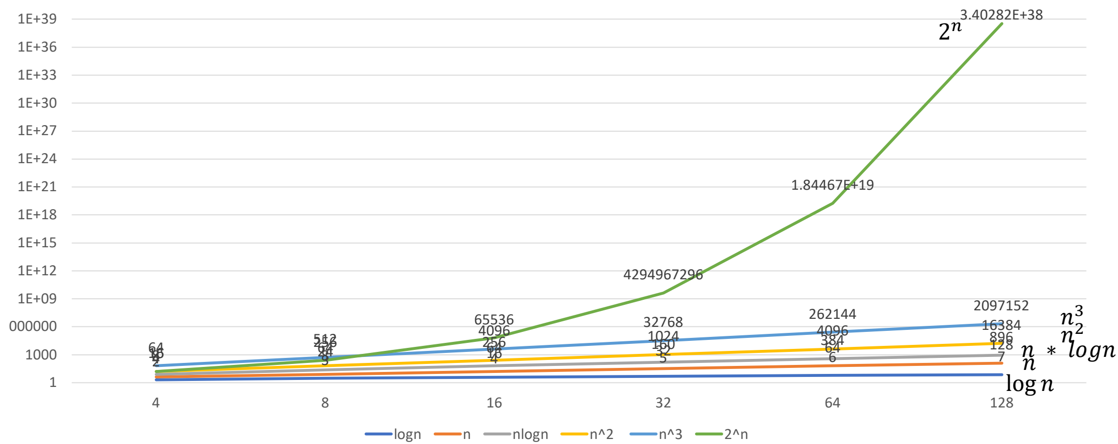
- Big-Oh gives us an upper bound on the number of operations
  - Most of the time, we are interested in Big-Oh complexity
- Big-Omega gives us the lower bound on the number of operations
  - Lower bound is useful in cases when some approximation is needed to compute the upper bound
- If both lower and upper bounds are the same, we use Big-Theta notation
  - Big-theta gives us the precise complexity that can’t be further reduced

Growth of functions



For n = 128, an algorithm with time complexity O(n) does around a multiple of 128 operations, whereas the same problem can be solved using an O(log n) algorithm in a multiple of 7 operations. If the complexity is O(n^3), it takes around a multiple of 2097152 operations to solve the problem. Therefore, reducing the complexity of an algorithm is a much better optimization than improving the hardware, for example. Improving the hardware may make your program 2x or 4x faster; however, reducing the complexity from n^2 to "n * log n" may make your program 10000 times better for a large input.

Growth of functions



For n=128, an O(2^n) algorithm performs around ($3.4 * 10^{38}$) operations, which may take a very long time. Therefore, O(2^n) algorithms are not an acceptable solution for many problems.

Problem size (n) vs Time complexity

Complexity	1 second	1 minute	1 hour
n	60M	360M	21,600M
n * log n	2M	113M	5,740M
n^2	7,746	60,000	4,64,758
n^3	392	1534	6000
2^n	26	32	38

Common complexities

Best

Complexity	Common name
O(1)	Constant
O(log(n))	Logarithmic
O(n)	Linear
O(nlog(n))	n-log-n
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential

Worst

Recurrence relation

Recurrence relation

- A recurrence relation is an equation or inequality that expresses the n^{th} term of a sequence as a function of the k preceding terms
  - k is called the order of the recurrence relation
  - e.g., `f(n) = f(n-1) + f(n-2)` is a recurrence relation

- Source: Wikipedia

Solving recurrence relation

- Expansion method
  - Also look at recursion tree method
- Intelligent guesses
  - Read from CLRS and Narasimha book
- Elimination of history
  - Full-history recurrence relation
    - Depends on all preceding terms instead of a few
  - We will discuss later in this course

Expansion method

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

$n>1$

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

$T(n) = T(n-2) + c + c = \underline{T(n-2)} + 2c$

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

$= \ldots$

$= T(n-k) + kc$

Substitute: $n-k=1$

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

The expansion method has four steps: 1> expand and simplify the recurrence a few times until you see a pattern, 2> compute the recurrence relation after k number of expansions, 3> substitute k with a value such that the RHS can be expressed in terms of base cases (i.e., T(1) in this example), and 4> simplify the equation to compute the final result. These steps are shown in more detail in the following slides.

Expansion method

- STEP-1 : Expand and simplify the recurrence

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

T(n-1) = T(n-2) + c
```

Expansion method

- STEP-1 : Expand and simplify the recurrence

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

T(n-1) = T(n-2) + c
Expand T(n) substituting the value of T(n-1)
T(n) = (T(n-2) + c) + c
```

Expansion method

- STEP-1 : Expand and simplify the recurrence

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

T(n-1) = T(n-2) + c
Expand T(n) substituting the value of T(n-1)
T(n) = (T(n-2) + c) + c
     = T(n-2) + 2c          // simplify
```

Expansion method

- STEP-1 : Expand and simplify the recurrence

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

T(n-1) = T(n-2) + c
Expand T(n) substituting the value of T(n-1)
T(n) = (T(n-2) + c) + c
     = T(n-2) + 2c          // simplify
Expand T(n) substituting the value of T(n-2)
T(n) = (T(n-3) + c) + 2c
```

Expansion method

- STEP-1 : Expand and simplify the recurrence

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

T(n-1) = T(n-2) + c
Expand T(n) substituting the value of T(n-1)
T(n) = (T(n-2) + c) + c
     = T(n-2) + 2c          // simplify
Expand T(n) substituting the value of T(n-2)
T(n) = (T(n-3) + c) + 2c
     = T(n-3) + 3c          // simplify
```

Expansion method

- STEP-2: Identify a pattern after the k^{th} iteration

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

T(n-1) = T(n-2) + c
Expand T(n) substituting the value of T(n-1)
T(n) = (T(n-2) + c) + c
     = T(n-2) + 2c          // simplify
Expand T(n) substituting the value of T(n-2)
T(n) = (T(n-3) + c) + 2c
     = T(n-3) + 3c          // simplify
. . . .
After expanding and simplifying k times
T(n) = T(n-k) + kc
```

Expansion method

- STEP-3: Substitute k to express T(n) as a function of base cases

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

After expanding and simplifying k times
T(n) = T(n-k) + kc

Substitute n-k = 1
T(n) = T(1) + c(n-1)
```

Expansion method

- STEP-4: Simplify the equation

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

After expanding and simplifying k times
T(n) = T(n-k) + kc

Substitute n-k = 1
T(n) = T(1) + c(n-1)
     = 1 + cn − c
```

Time complexity of recursive programs

Time complexity

- For recursive programs, time complexities can be expressed as recurrence relations

Factorial

```
1. int factorial(int n) {
2.     if (n == 0)
3.         return 1;
4.     return n * factorial(n-1);
5. }
```

Time complexity:

$T(0) = 2$

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

$T(n) = T(n-2) + c + c = T(n-2) + 2c$

$\quad = T(n-3) + c + 2c = T(n-3) + 3c$

$\quad \ldots$

$\quad = T(n-k) + kc$

Subtitute, $n - k = 0$

$T(n) = T(0) + n \times c$

$\quad = 2 + n \times c$

$\quad = O(n)$

If n=0, the factorial function executes two operations. You can also consider it some constant “c” as it won’t change the complexity. Now factorial(n) calls factorial(n-1) at line-4. Because factorial(n) performs T(n) operations, factorial(n-1) will perform T(n-1) operations. Apart from the T(n-1) operations corresponding to the recursive call at line-4, factorial(n) additionally performs a constant number of operations, say c. Therefore, the total number of operations performed by factorial(n). i.e., T(n), is T(n-1) + c. Solving this recurrence relation will give us the desired time complexity.

Time complexity

$T(0) = 2$

$$
\begin{aligned}
T(n) &= T(n-1) + c \\
&= (T(n-2) + c) + c = T(n-2) + 2c \\
&= (T(n-3) + c) + 2c = T(n-3) + 3c \\
&= \cdots \\
&= T(n\ - k) + c\mathrm{k}
\end{aligned}
$$

Substitute, k = n

$$
\begin{aligned}
&= T(0) + cn = 2 + \mathrm{cn} \\
&= O(n)
\end{aligned}
$$

Linear search

```
1. int lsearch(int arr[], int val, int n) {
2.    if (n == 0)
3.       return -1;
4.    if (arr[n-1] == val)
5.       return n-1;
6.    return lsearch(arr, val, n-1);
7. }
```

Time complexity:

$T(0) = 2$

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

$O(n)$

If n=0, lsearch performs 2 (or some constant) operations. Because we are interested in worst-case time complexity, let’s say the comparison at line-4 never holds. Let’s assume for n > 0, lsearch(arr, val, n) does T(n) operations, where n is the input size. Therefore, lsearch(arr, val, n-1) will execute T(n-1) operations because the input size is n-1. Apart from calling lsearch(arr, val, n-1), lsearch(arr, val, n) also does some constant number of operations, say c. Therefore, the total number of operations performed by lsearch, i.e., T(n), is T(n-1) + c, when n > 0. Solving this recurrence relation gives us the desired time complexity.

Binary search

$hi - lo + 1$

$hi = n-1$
0

```
1. int bsearch(int arr[], int val, int lo, int hi) {
2.    if (hi < lo)
3.       return -1;
4.    int mid = (lo + hi) / 2;
5.    if (arr[mid] == val)
6.       return mid;
7.    if (arr[mid] > val)
8.       return bsearch(arr, val, lo, mid-1);
9.    else
10.       return bsearch(arr, val, mid+1, hi);
11.}
```

Time complexity:

$T(1) = c$

$T(n) = T(\frac{n}{2}) + c_1$

$T(n) = T(\frac{n}{4}) + c_1 + c_1$

$= T(\frac{n}{2^2}) + 2c_1$

$= T(\frac{n}{2^3}) + c_1 + 2c_1$

$= T(\frac{n}{2^3}) + 3c_1$

$= \dots$

$= T(\frac{n}{2^k}) + kc_1$

Substitute: $\frac{n}{2^k} = 1 \Rightarrow k = \log_2 n$

$T(n) = T(1) + c_1 * \log_2 n$

$= c + c_1 * \log_2 n$

$O(\log n)$

Let's analyze the bsearch algorithm. Here, lo is the index of the first element, hi is the index of the last element, and the number of elements or the input size is (hi – lo + 1). If the input size is 1, then this algorithm is going to execute a constant number of operations (try to compute the number of operations using the method we discussed in class if you are not convinced). Otherwise, if the "if-branch" is taken bsearch is called at line-8 with input size (mid -lo) = n/2, else in the "else-branch" bsearch is called at line-10 with the input size (hi-mid) = n/2. Both of these recursive calls will execute T(n/2) operations each if bsearch with input n performs T(n) operations. Because, at a given time, either the if or the else branch is taken, and the cost of both the branches are the same, the recurrence relation corresponding to the time-complexity of the bsearch algorithm is T(n) = T(n-1) + c_1, where c_1 is the additional constant number of operations performed by the bsearch algorithm apart from calling the bsearch routines at lines-8,10. When we use the expansion method to solve the recurrence relation, after the k-th expansion, T(n) is equal to T(n/2^k) + (k * c_1). Substituting n = 2^k gives us the time complexity O(log n).

Time complexity

$T(1) = c$

$$T(n) = T\left(\frac{n}{2}\right) + c_1$$
$$= (T\left(\frac{n}{2^2}\right) + c_1) + c_1 = T\left(\frac{n}{2^2}\right) + 2c_1$$

$$= \left(T\left(\frac{n}{2^3}\right) + c_1\right) + 2c_1 = T\left(\frac{n}{2^3}\right) + 3c_1$$
$$= \ \dots$$

$$= T\left(\frac{n}{2^k}\right) + kc_1$$

Substitute $n = 2^k$
$T(n) = T(1) + (log_2\, n) * c_1 = O(\log(n))$

O(log n) vs O(n)

O(log n) vs O(n)

- An algorithm is O(log n) if it takes constant time to cut the problem size by a fraction
  - Usually 1/2 as in the binary search algorithm or the fast algorithm for power
  - The power algorithm that divides the problem into 1/3 is also O(log n)

- An algorithm is O(n) if it takes constant time to reduce the problem size by a constant amount
  - E.g., linear search reduces the problem size by one in a constant number of steps

gcd

gcd

```
int gcd(int m, int n) {
  int rem;
  while (n > 0) {
    rem = m % n;
    m = n;
    n = rem;
  }
  return m;
}
```

```
gcd(21, 13)
```

13) 21 (1
13
8) 13 (1
8
5) 8 (1
5
3) 5 (1
3
2) 3 (1
2
1) 2 (2
2
0

15) 26 (1
15

gcd

```
1. int gcd(int m, int n) {
2.   int rem;
3.   while (n > 0) {
4.     rem = m % n;
5.     m = n;
6.     n = rem;
7.   }
8.   return m;
9. }
```

How many iterations are needed to break the problem into 1/2

if n <= m/2

After 1 iteration m will be at most $m/2$

if $n > m/2$

After 1 iteration $m = n$ therefore $m > \frac{m}{2}$.

After 1 iteration $n = rem$ and $rem < \frac{m}{2}$ therefore $n < \frac{m}{2}$

After 2nd iteration because m takes the prev value of n $m < m/2$

If we look at this algorithm closely, if n <= m/2 after the first iteration, the value of m will be <= m/2, because of the assignment m = n, at line-5. On the other hand, if n > m/2, the value of n after the first iteration would be < m/2. Because the remainder computed at line-4, m % n will be less than m/2, when n > m/2 (rem will be m – n, which is less than m/2). During the second iteration, the value of n from the first iteration is assigned to m at line-5. Therefore, after the second iteration, the value of m will be less than half of the value of m before the first iteration. From this, we can conclude that in two iterations, the value of m will definitely be reduced by half of its original value.

gcd

```
int gcd(int m, int n) {
  int rem;
  while (n > 0) {
    rem = m % n;
    m = n;
    n = rem;
  }
  return m;
}
```

Time complexity

$m \quad \frac{m}{2} \quad \frac{m}{4} \quad \frac{m}{8} \quad \ldots \quad 1$

iterations $\quad 2 \quad 2 \quad 2 \quad \ldots \quad 2$

$\frac{m}{2} \quad \frac{m}{2^2} \quad \frac{m}{2^3} \quad \ldots \quad \frac{m}{2^k}$

$\frac{m}{2^k} = 1$

$k = \log_2 m$

This algorithm iterates at most $2 \log_2 m$ times.

$2 \log_2 n + 1$ iterations

$O(\log n)$

Notice, in the worst case, the loop runs until "m" reaches one. So, the value of "m" will be m/2 after two iterations, $m/2^2$ after four iterations, $m/2^3$ after six iterations, and so on. Therefore after 2k iterations, the value of m will be $m/2^k$. Let's say 2k is the number of iterations in the worst case; so, the value of m after 2k iterations, i.e., $m/2^k$, will be equal to 1. This gives the worst-case number of iterations as $(2 * \log_2 m)$. The number of iterations in terms of n will be $(2 * \log_2 n + 1)$ because, after the first iteration, m becomes n. Since every iteration does a constant number of operations, the worst-case time complexity of the gcd algorithm is O(log n).

gcd

```
1. int gcd(int m, int n) {
2.   int rem;
3.   while (n > 0) {
4.     rem = m % n;
5.     m = n;
6.     n = rem;
7.   }
8.   return m;
9. }
```

```
How many iterations are needed to break the
problem into 1/2

if n <= m/2
 first iteration
  rem < m/2  // at line-4
  m <= m/2   // at line-5
  n < m/2    // at line-6

if n > m/2
 first iteration
  rem < m/2  // at line-4
  m > m/2    // at line-5
  n < m/2    // at line-6

 second iteration
  rem < m/2 (because n < m/2) // at line-4
  m < m/2 (because n < m/2)   // at line-5
  n < m/2                     // at line-6
```

gcd

```
int gcd(int m, int n) {
  int rem;
  while (n > 0) {
    rem = m % n;
    m = n;
    n = rem;
  }
  return m;
}
```

```
Problem size:  m m/2 m/4 m/8  … 1
Num iterations:   2   4   6     2k
```

After 2k iterations m becomes $m/2^k$
If the total number of iterations in the worst case is 2k:

$$\frac{m}{2^k} = 1$$

$$k = \log_2 m$$

Therefore, number of iterations in the worst case is: 2 * $\log_2 m$
or 1 + 2 * $\log_2 n$
because after the first iteration m becomes n

Time complexity: O(log n)

Selection sort

```
1. void selection_sort(int arr[], int pos, int n)
2.  {
3.   int idx;

4.   if (pos >= n - 1) {
5.     return;
6.   }

7.   idx = find_min(arr, pos, n);
8.   if (idx != pos) {
9.     swap(arr, pos, idx);
10.   }
11.   selection_sort(arr, pos+1, n);
12.}
```

```
int find_min(int arr[], int start, int end)
{
  int i, min, idx;

  min = arr[start];
  idx = start;
  for (i = start + 1; i < end; i++) {
    if (arr[i] < min) {
      min = arr[i];
      idx = i;
    }
  }
  return idx;
}

void swap(int arr[], int i, int j)
{
  int tmp = arr[i];
  arr[i] = arr[j];
  arr[j] = tmp;
}
```

Time complexity:

$T(n) = T(n-1) + c_1 n + c_2$

Let's look at the selection sort algorithm. Initially, the value of pos is zero, and n is the total number of elements. The input size is n – pos. At line-11, selection_sort is called with the input size "n – pos – 1", which is one less than the current input size. If the selection_sort performs T(n) operations, the recursive call at line-11 will perform T(n-1) operations. Besides the recursive call, find_min iterates (end-start-1) times, i.e., (n-pos-1) times, roughly equal to the input size. If n is the input size, find_min performs roughly $c_1*n + c_3$ operations. In addition to the number of operations in find_min and the recursive call, selection_sort also executes a constant number of operations, say c_4. So, the total number of operations by selection sort can be expressed as $T(n) = T(n-1) + c_1*n + c_2$, where $c_2 = c_3+c_4$.

Time complexity

$T(1) = c$

$T(n) = T(n-1) + c_1 n + c_2$

$= \left(T(n-2) + c_1(n-1) + c_2\right) + c_1 n + c_2$

$= T(n-2) + c_1(n + (n-1)) + 2c_2$

$= \left(T(n-3) + c_1(n-2) + c_2\right) + c_1(n + (n-1)) + 2c_2$

$= T(n-3) + c_1(n + (n-1) + (n-2)) + 3c_2$

$\ldots$

$= T(n-k) + c_1(n + (n-1) + (n-2) + \ldots + (n-k+1)) + kc_2$

Substitute $n - k = 1 \rightarrow k = n-1$

$= T(1) + c_1(n + (n-1) + (n-2) + \ldots + 2) + (n-1)c_2$

$= c + c_1(1 + 2 + \ldots + n - 1) + (n-1)c_2$

$= c + c_1\left(\frac{n(n+1)}{2} - 1\right) + (n-1)c_2$

$O(n^2)$

After solving this recurrence relation using the expansion method, we obtain the time complexity of selection_sort as $O(n^2)$.

Time complexity

$T(1) = c$
$T(n) = T(n-1) + c_1 n + c_2$

$$= (T(n-2) + c_1(n-1) + c_2) + c_1 n + c_2$$
$$= T(n-2) + c_1(n-1) + c_1 n + 2c_2$$
$$= (T(n-3) + c_1(n-2) + c_2) + c_1(n-1) + c_1 n + 2c_2$$
$$= T(n-3) + c_1(n-2) + c_1(n-1) + c_1 n + 3c_2$$
$$= \ \ldots$$
$$= T(n-k) + c_1(n-k+1) + \cdots + c_1(n-2) + c_1(n-1) + c_1 n + kc_2$$

Substitute, k = n-1

$$= T(1) + 2c_1 + \ \ldots + c_1(n-2) + c_1(n-1) + c_1 n + c_2(n-1)$$
$$= c + c_1(2 + 3 + \cdots + n) + c_2(n-1)$$
$$= c + c_1\left(\frac{n(n+1)}{2} - 1\right) + c_2(n-1)$$
$$= O(n^2)$$

Towers of Hanoi

```
1. void move(int n, char src_t[], char dst_t[], char tmp_t[]) {
2.   if (n == 1) {
3.     printf(“moving from %s to %s\n”, src_t, dst_t);
4.   }
5.   else {
6.     move(n-1, src_t, tmp_t, dst_t);
7.     printf(“moving from %s to %s\n”, src_t, dst_t);
8.     move(n-1, tmp_t, dst_t, src_t);
9.   }
10.}

11.int main() {
12.  move(4, “Tower1”, “Tower3”, “Tower2”);
13.  return 0;
14.}
```

Time complexity:

$T(1) = c$

$T(n) = 2T(n-1) + c_1$

Let’s look at the Towers of Hanoi problem. In this case, if the move routine executes T(n) operations, the recursive calls at lines-6,7 will execute T(n-1) operations each. The move routine additionally executes a constant number of operations (say c_1) besides the recursive calls. In this case, the recurrence relation for the time complexity is $T(n) = 2T(n-1) + c_1$.

Time complexity

$1 + r + r^2 + \ldots + r^{n-1} = \frac{r^n - 1}{r - 1}$

$T(1) = c$

$$T(n) = 2T(n-1) + c_1$$
$$= 2(2T(n-2) + c_1) + c_1 = 2^2 \underline{T(n-2)} + c_1(1+2)$$
$$= 2^2(2T(n-3) + c_1) + c_1(1+2) = 2^3 T(n-3) + c_1(1+2+2^2)$$
$$= \ldots$$
$$= 2^k T(n-k) + c_1(1+2+2^2+\ldots+2^{k-1})$$

Substitute: $n - k = 1 \rightarrow k = n-1$

$$= 2^{n-1} T(1) + c_1(2^k - 1)$$
$$= 2^{n-1} \times c + c_1(2^{n-1} - 1)$$

$O(2^n)$

If we solve this recurrence relation using the expansion method, we obtain the time complexity as $O(2^n)$. It means that the time taken by this function grows exponentially, and it might take a lot of time for large inputs.

Time complexity

$$
\begin{aligned}
T(1) &= c \\
T(n) &= 2T(n-1) + c_1 \\
&= 2(2T(n-2) + c_1) + c_1 \\
&= 2^2T(n-2) + 2c_1 + c_1 \\
&= 2^2(2T(n-3) + c_1) + 2c_1 + c_1 \\
&= 2^3T(n-3) + 2^2c_1 + 2c_1 + c_1 \\
&= \ \ldots \\
&= 2^kT(n-k) + 2^{k-1}c_1 + \ \ldots + 2^2c_1 + 2c_1 + c_1
\end{aligned}
$$

Substitute n - k = 1

$$
\begin{aligned}
&= 2^{n-1}T(1) + 2^{n-2}c_1 + \ \ldots + 2^2c_1 + 2c_1 + c_1 \\
&= c * \ 2^{n-1} + c_1(1 + 2 + \cdots + 2^{n-2}) \\
&= c * \ 2^{n-1} + c_1(2^{n-1} - 1) \\
&= O(2^n)
\end{aligned}
$$

Master theorem

A recurrence relation of the form

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k, where\ a \geq 1, b \geq 2, c \geq 0, k \geq 0$$

has the following solution

$$T(n) = \begin{cases} O(n^{\log_b a}) & if\ a > b^k \\ O(n^k logn) & if\ a = b^k \\ O(n^k) & if\ a < b^k \end{cases}$$

We can also use the master theorem to solve a subset of recurrence relations of the form shown on this slide.

Useful formulae

$$1 + 2 + 3 + \ \dots + n = \frac{n(n+1)}{2}$$

$$1 + r + r^2 + \ \dots + r^{n-1} = \frac{r^n - 1}{r - 1} \qquad if\ r \neq 1$$

$$1^2 + 2^2 + 3^2 + \ \dots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$1 + \frac{1}{2} + \frac{1}{3} + \ \dots + \frac{1}{n} \approx \ln x$$

Useful formulae

$$\log_b a = \frac{1}{\log_a b}$$

$$\log_a x = \frac{\log_b x}{\log_b a}$$

$$b^{\log_b x} = x$$

$$b^{\log_a x} = x^{\log_a b}$$

Useful formulae

$$f(x) = ax^2 + bx + c \quad where\ a, b, c \in R\ and\ a\ \neq 0$$

The roots of the above quadratic equation are:

$$\alpha = \frac{-b+\sqrt{b^2-4ac}}{2a} \qquad \beta = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

Fibonacci numbers

```
int fib(int n) {
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

$$T(n) = T(n-1) + \underline{T(n-2)} + c$$
$$\leq 2T(n-1) + c$$

$$T(n) \geq 2T(n-2) + c$$

Time complexity:

$T(0) = 2$
$T(1) = 2$

$$T(n) = T(n-1) + T(n-2) + c$$
$$= T(n-2) + T(n-3) + c + T(n-3) + T(n-4) + c + c$$
$$= T(n-2) + 2T(n-3) + T(n-4) + 3c$$
$$= T(n-3) + T(n-4) + c + 2(T(n-4) + T(n-5) + c) + T(n-5) + T(n-6) + c + 3c$$
$$= T(n-3) + 3T(n-4) + 3T(n-5) + T(n-6) + 7c$$

Let's return to the Fibonacci numbers we have been chasing since the beginning. Notice that if we use the expansion method, it's not going to work in this case because there is no pattern. For such problems, we can make some approximation to convert this into a recurrence relation that is solvable using the expansion method. For the upper bound, we can over-approximate the number of operations required to compute the result. An over-approximation that works, in this case, is to replace fib(n-2) with fib(n-1), which gives us the recurrence relation, fib(n) <= 2 * fib(n-1) + c. For the lower bound, we can under-approximate the total number of operations. An under-approximation that works, in this case, is to replace fib(n-1) with fib(n-2), yielding a relation, fib(n) >= 2 * fib(n-2) + c. The upper bound gives us the Big-Oh complexity, and the lower bound gives us the Big-Omega complexity; both are needed in this case because finding the actual complexity is slightly more complicated and requires more sophisticated analysis.

Fibonacci numbers

```
int fib(int n) {
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

Time complexity:

Upper Bound:
T(1) = 2
T(0) = 2

T(n) <= 2T(n-1) + c

Fibonacci numbers

```
int fib(int n) {
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

Time complexity:

Lower Bound:
T(1) = 2
T(0) = 2

T(n) >= 2T(n-2) + c