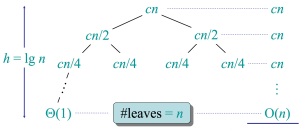
$T(n)$
$\Theta(1)$
$2T(n/2)$
$O(n)$

MERGE-SORT $A[1\,..\,n]$

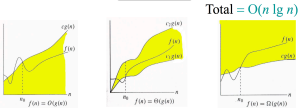
1. If $n = 1$, done.
2. Recursively sort $A[\,1\,..\lceil n/2 \rceil]$ and $A[\,\lceil n/2 \rceil + 1\,..\,n\,]$.
3. *"Merge"* the 2 sorted lists

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + O(n) \text{ if } n > 1. \end{cases}$$

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Total = $O(n \lg n)$



If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \text{then } f(n) \text{ is } O(g(n)) \\ c > 0 & \text{then } f(n) \text{ is } \Theta(g(n)) \\ \infty & \text{then } f(n) \text{ is } \Omega(g(n)) \end{cases}$

$\lim_{n\to\infty} 2^n/3^n = \lim_{n\to\infty} (2/3)^n$

Now we use the following Theorem

$$\lim_{n\to\infty} \alpha^n = \begin{cases} 0 & \text{if } \alpha < 1 \\ 1 & \text{if } \alpha = 1 \\ \infty & \text{if } \alpha > 1 \end{cases}$$

$\lim_{n\to\infty} 2^n/3^n = \lim_{n\to\infty} (2/3)^n = 0$

We set up our limit

$\lim_{n\to\infty}$ f(n)/g(n) = $\lim_{n\to\infty} \log_2 n / \log_3 n^2$
= $\lim_{n\to\infty} \log_2 n / (2\log_2 n)$

Here we use the change of base formula for logarithms:

$\log_3 n = \log_2 n \,/\, \log_2 3$
$\lim_{n\to\infty} \log_2 n / (2\log_2 n) = \lim_{n\to\infty} (\log_2 n \log_2 3) / (2\log_2 n)$
= $\lim_{n\to\infty} (\log_2 3)/2$
= $(\log_2 3)/2$
≈ 0.7924, a positive constant

So we conclude that f(n)∈Θ(g(n)) that is $\log_2 n \in \Theta(\log_3 n^2)$ which implies that $\log_3 n^2 \in \Theta(\log_2 n)$

Theorem:

$f(n) = \Theta(g(n)) \Leftrightarrow f = O(g(n))$ and $f = \Omega(g(n))$

- Transitivity:
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  - Same for O and Ω
- Reflexivity:
  - $f(n) = \Theta(f(n))$
  - Same for O and Ω
- Symmetry:
  - $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- Transpose symmetry:
  - $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

Transitivity $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$

1. By definition $f(n) = \Theta(g(n))$ implies there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1\, g(n) \le f(n) \le c_2\, g(n)$ for all $n \ge n_0$

2. By definition $g(n) = \Theta(h(n))$ implies there exist positive constants c_3, c_4, and n_1 such that $0 \le c_3\, h(n) \le g(n) \le c_4\, h(n)$ for all $n \ge n_1$

3. Show $f(n) = \Theta(h(n))$ that is there exist positive constants c_5, c_6, and n_2 such that $0 \le c_5\, h(n) \le f(n) \le c_6\, h(n)$ for all $n \ge n_2$

By combining 1 and 2: $c_1 c_3\, h(n) \le c_1\, g(n) \le f(n)$ let $c_5 = c_1 c_3$ so $c_5\, h(n) \le f(n)$
Again from 1 and 2: $f(n) \le c_2\, g(n) \le c_2 c_4\, h(n)$ let $c_6 = c_2 c_4$ so $c_6\, h(n) \le f(n)$
And let $n_2 = \max\{n_0, n_1\}$

pseudocode

```
findIfTwoSum(S, x):
    sorted = mergeSort(S)                                  1 * O(n lg n)   -> O(n lg n)
    for i = 0 to sorted.length                             n * O(1)        -> O(n)
        compliment = x - sorted[i]                         n * O(1)        -> O(n)
        if (binarySearch(sorted, compliment) != -1)        n * O(lg n)     -> O(n lg n)
            return True                                    1 * O(1)        -> O(1)
    return False                                           1 * O(1)        -> O(1)
```

times executed * cost -> *total cost*

There are two steps in the algorithm with cost of O(n lg n), resulting in a total cost of 2 n lg n; however, constant factors are discounted in asymptotic analysis; therefore, the total cost of the algorithm is still O(n lg n).

Algorithm *mergeSort*(*S*, *c*)

Input sequence ***S*** with ***n*** elements, comparator ***c***
Output sequence ***S*** sorted according to ***c***
if ***S.size*()** > 1
  (S_1, S_2) ← ***partition***(***S***, ***n***/2)
  mergeSort(S_1, ***c***)
  mergeSort(S_2, ***c***)
  S ← ***merge***(S_1, S_2)

```
long power (long x, long n) {
   if(n == 0)
        return 1;
   else if(n == 1)
        return x;
   else if ((n % 2) == 0){
        temp = power(x, n/2);
        return temp*temp;
   }
   else {
        temp = power(x,(n-1)/2)
        return x * temp* temp;
   }
}
```

```
long fibonacci (int n) {
// Recursively calculates Fibonacci number
  if( n == 0)
      return 0;
  else if( n == 1)
      return 1;
  else
      return fibonacci(n - 1) + fibonacci(n - 2);
}
```

```
Hanoi(n, from, to, temp){
   if (n == 1)
      Move(from, to);
   else{
      Hanoi(n - 1, from, temp, to);
      Move(from, to);
      Hanoi(n - 1, temp, to, from);
   }
}
```

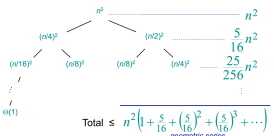
T(n) = c + T(n/2)

T(n) = c + T(n/2) = c + c + T(n/4) = c + c + c + T(n/8)
T(n/2) = c + T(n/4)
T(n/4) = c + T(n/8)

Stop when $n/2^i = 1$ => $i = \lg n$

T(n) = c + c + … + c + T(1) (c repeated n times)
= clgn + T(1)
= Θ(lgn)

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Master Method

"Formula" for solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, a ≥ 1, b > 1, and f(n) > 0

case 1: if $f(n) = O(n^{\log_b a - \varepsilon})$ for some ε > 0, then: $T(n) = \Theta(n^{\log_b a})$

case 2: if $f(n) = \Theta(n^{\log_b a})$, then: $T(n) = \Theta(n^{\log_b a} \lg n)$

case 3: if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some ε > 0, and if
$af(n/b) \le cf(n)$ for some c < 1 and all sufficiently large n, then:
$T(n) = \Theta(f(n))$

Master Method – Example 1

T(n) = 2T(n/2) + n² a = 2, b = 2, $\log_2 2 = 1$

compare n with f(n) = n²

case 3: if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some ε > 0

⇒ $f(n) = \Omega(n^{1+\varepsilon})$ case 3 ⇒ verify regularity cond.

a f(n/b) ≤ c f(n)

⇔ 2 n²/4 ≤ c n² ⇒ c = ½ is a solution (c<1)

⇒ $T(n) = \Theta(n^2)$

Decrease and Conquer

Muster Method for *"decrease and conquer"* recurrences of the form

$$T(n) = a\,T(n-b) + f(n)$$

for some integer constants a, b > 0, d ≥ 0 .

If $f(n)$ is $O(n^d)$ then

$$T(n) = \begin{cases} O(n^d), & \text{if } a < 1, \\ O(n^{d+1}), & \text{if } a = 1 \\ O(n^d a^{n/b}), & \text{if } a > 1. \end{cases}$$

Decrease and Conquer: Example

Tours of Hanoi recurrence: T(n) = 2 T(n-1) + 1

By the Muster Method: T(n) = a T(n-b) + f(n)

a = 2, b= 1, f(n) = 1 so d = 0. f(n) = $\Theta(n^0)$

$$T(n) = \begin{cases} \Theta(n^d), & \text{if } a < 1, \\ \Theta(n^{d+1}), & \text{if } a = 1 \\ \Theta(n^d a^{n/b}), & \text{if } a > 1. \end{cases}$$

Therefore, T(n) is $\Theta(n^0 2^{n/1}) = \Theta(2^n)$.

```
fib (n) {
  if (n = 0) {
     return 0;
  } else if (n = 1) {
     return 1;
  } else {
     return fib(n-1) + fib(n-2);
  }
}
```

```
memo = { }
fib (n) {
  if (n in memo) { return memo[n] }
  if (n <= 1) {
     f = n;
  } else {
     f = fib(n-1) + fib(n-2);
  }
  memo[n] = f;
  return f
}
```

```
fib = { }
fib[0] = 0;
fib[1] = 1;
for k = 2 to n
      fib[k] =  fib[k-1] + fib[k-2];
return fib[n]
```

```
/** using divide-and-conquer
 * Calculates n choose k
 *   n the total number to choose from (n > 0)
 *   k the number to choose (0 <= k <= n)
 * */
int Binomial(int n, int k) {
   if (k == 0 || n == k) {
      return 1;
   }
   else {
      return Binomial(n-1, k-1) + Binomial(n-1, k);
   }
}
```

Binomial (n,k)
// Computes C(n,k) by DP
// Input: A pair of nonnegative integers n≥k≥0
// Output: the value of C(n,k)

```
for i ← 0 to n do
    for j ← 0 to min(i, k) do
        if j = 0 or j = i
            C[i, j] ← 1
        else    C[i, j] ← C[i-1, j-1] + C[i-1, j]
Return C[n.k]
```

LCS-Length(X, Y)

```
1. m = length(X)  // get the # of symbols in X
2. n  = length(Y) // get the # of symbols in Y
3. for i = 1 to m      c[i,0] = 0     // special case: Y0
4. for j = 1 to n      c[0,j] = 0     // special case: X0
5. for i = 1 to m                     // for all Xi
6.       for j = 1 to n               // for all Yj
7.            if ( Xi == Yj )
8.                 c[i,j] = c[i-1,j-1] + 1
9.            else c[i,j] = max( c[i-1,j], c[i,j-1] )
10. return c
```

```
if i = 0 or j = 0                    Running time: Θ(m + n)
  then return
if b[i, j] = "↖"
  then PRINT-LCS(b, X, i - 1, j - 1)
       print xi
elseif b[i, j] = "↑"
       then PRINT-LCS(b, X, i - 1, j)
       else PRINT-LCS(b, X, i, j - 1)
```

0-1 Knapsack Algorithm

```
for w = 0 to W
   B[0,w] = 0
for i = 0 to n
   B[i,0] = 0
   for w = 0 to W
      if wi <= w // item i can be part of the solution
            if bi + B[i-1,w-wi] > B[i-1,w]
                  B[i,w] = bi + B[i-1,w- wi]
            else
                  B[i,w] = B[i-1,w]
      else B[i,w] = B[i-1,w]  // wi > w
```

Formula

$coins[i] = inf$ if $i < 0$
$coins[0] = 0$
$coins[1] = 1$

$$coins[j] = \min_{i} \{ 1 + coins[j - v_i] \}$$

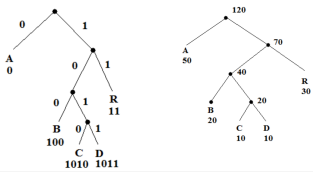
$1 \le i \le n$

```
minCoins(A, v[], n)
      coins[0]=0; coins[1] = 1;
      for j = 2 to A do {
            min = inf
            for i = 1 to n do {
                  if ( j >= v[i] )
                        if (coins[j-v[i]] < min )
                              min = coins[j-v[i]]
            }
            coins[j] = min + 1;
      }
      return coins[A]
```

Algorithm *FKnapsack*(*S*, *W*)

Input: set ***S*** of items w/ benefit b_i and weight w_i; max. weight ***W***
Output: amount x_i of each item ***i*** to maximize benefit with weight at most ***W***
for *each item i in S*
  $x_i \leftarrow 0$
  $v_i \leftarrow b_i / w_i$ {value}
w ← 0 {current total weight}
while *w* < *W*
  remove item ***i*** with highest v_i
  $x_i \leftarrow \min\{w_i, W - w\}$
  $w \leftarrow w + \min\{w_i, W - w\}$

```
long power (long x, long n) {
   if(n == 0)
     return 1;
   else if(n == 1)
        return x;
   else if ((n % 2) == 0)
    return power (x, n/2) * power (x, n/2);
   else
    return x * power (x,(n-1)/2) * power (x,(n-1)/2);
}
```



GREEDY-ACTIVITY-SELECTOR(s, f)

```
1  n ← length[s]
2  A ← {a1}
3  i ← 1
4  for m ← 2 to n
5       do if sm ≥ fi
6             then A ← A ∪ {am}
7                  i ← m
8  return A
```

Algorithm TaskSchedule(*T*)

Input: set ***T*** of tasks w/ start time s_i and finish time f_i
Output: non-conflicting schedule with minimum number of machines
m ← 0 {no. of machines}
while *T is not empty*
  remove task i w/ smallest s_i
  if *there's a machine j for i* then
    schedule i on machine j
  else
    m ← ***m*** + 1
    schedule i on machine m

Suppose you have a set of classes to schedule among a large number of lecture halls, where any class can class place in any lecture hall. Each class c_i has a start time s_i and finish time f_i. We wish to schedule all classes using as few lecture halls as possible. Verbally describe an efficient greedy algorithm to determine which class should use which lecture hall at any given time. What is the running time of your algorithm?

```
Algorithm CourseScheduler(C):
    Input: A set C of n courses, such that each course has a start time s_i and a finish time f_i
    Output: A nonconflicting schedule of courses in C using a minimum number lecture halls L.
    m= 0      Optimal number of lecture halls.
    C <- a heap with earliest start time at top of heap
    L <- empty heap
    While (heap C not empty)
        remove from C the course i with earliest start time s_i
            if there is a lecture hall j, 1 <= j <= m, with no course conflicting with course i  then
                schedule course i in lecture hall  j
            else
                m = m + 1
                schedule course i on lecture hall m
```

The key here is to use two heaps: The first, C, to represent the set of courses to be scheduled as shown in the augmented algorithm description above. The second, L, represents the lecture halls on which courses have been scheduled and the ending time of the last scheduled course on each lecture hall, as shown in the code below. The formation of the heap can be done in time n log n as was discussed with the heap data structure. The while loop is executed n times (once for each course added to the schedule). The removal of the top element from the heap takes time O(log n) and together this is O(nlgn). Sorting is Θ(nlgn)

The running time is O(nlgn) depending on the data structure used Θ(nlgn) and $O(n^2)$ are also acceptable answers.

2. You're going on a road trip with friends. Unfortunately, your headlights are broken, so you can only drive in the daytime. Therefore, on any given day you can drive no more than d miles. You have a map with n different hotels and the distances from your start point to each hotel $x_1 < x_2 < \ldots < x_n$. Your final destination is the last hotel. Describe an algorithm that determines which hotels you should stay in if you want to minimize the number of days it takes you to get to your destination.

The optimal strategy is the obvious greedy one. Beginning from the start point, we should go to the farthest hotel that we can get to within d miles. Stay there for one night. Then go to the farthest hotel we can get to within d miles of where stayed, and rest there, and so on.

If there are n hotels on the map, we need to inspect each one just once. The running time is therefore Θ(n).

3. **Scheduling jobs intervals with penalties**:

For each $1 \le i \le n$ job j_i is given by two numbers d_i and, where d_i is the deadline and p_i is the penalty. The length of each job is equal to 1 minute. We want to schedule all jobs, but only one job can run at any given time. If job i does not complete on or before its deadline d_i we should pay its penalty. Design a greedy algorithm to find a schedule which minimizes the sum of penalties.

Observation: We can assume that all jobs finish after n minutes. Suppose not. So there is an empty minute starting at say $0 \le k \le n-1$ (where no job is scheduled) and there is a job j_i for some $1 \le i \le n$ which is scheduled some time after n minutes. If we schedule job j_i to start at minute k, then this can only be a better solution since everything remains same except j_i might now be able to meet its deadline.

We assign time intervals M_i for $1 \le i \le n$ where M_i starts at minute $i - 1$ and ends at minute i. The greedy algorithm is as follows:

- Arrange the jobs in decreasing order of the penalties $p1 \ge p2 \ge \cdots \ge pn$ and add them in this order
- To add job j_i ,
  - if any time interval M_i is available for $1 \le i \le d_i$, then schedule j_i in the last such available interval.
  - else schedule j_i in the first available interval starting backwards from Mn

What is the running time of your algorithm? Explain.

1. The sorting takes Θ(nlgn)

2. To find the spot in the schedule can take in worst case 1+2+…+ n = $O(n^2)$ depending on the data structure used this could be reduced so you can use Big O instead of theta.

Overall $O(n^2)$.

1. (True/False) The running time of a dynamic programming algorithm is always Θ(P) where P is the number of subproblems.

Solution: False. The running time of a dynamic program is the number of subproblems times the time per subproblem. This would only be true if the time per subproblem is $O(1)$.

2. If f(n) = O(g(n)) and g(n) = O(h(n)), then f(n) = O(h(n)) True

3. Give asymptotic upper and lower bounds. Make as tight as possible.

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

Solution: By Case 3 of the Master Method, we have $T(n) = \Theta(n \lg n)$.

4. What does the fact given below imply regarding big-O, big-Ω and/or big-Θ relationships between the functions.

For all n > 40, 3g(n) ≤ f(n) ≤ 5g(n)

Solution: **f(n) is Θ(g(n)) which implies that g(n) is Θ(f(n))**

5. Order the following functions in increasing order of asymptotic (big-O) complexity.

$f(n) = 2^{2^{1000}}$, $g(n) = \sum_{i=1}^{n} (i+1)$, $h(n) = 2^n$, $p(n) = 10^{10} n$, $q(x) = n 2^{n/2}$

Solution:

f(n), p(n), g(n) q(n), h(n)

6. Show log(n!) is O(nlogn)

Solution:

log(n!) = log (n*(n-1)*(n-2)*…2*1)

= logn + log(n-1) + log(n-2) + .. + log 2 + log1 < nlogn

Therefore, O(nlogn)