Smart(er) searching and sorting

Alexandra Silva and Henk Barendregt

{alexandra,henk}@cs.ru.nl
http://www.cs.ru.nl/~{alexandra,henk}

Institute for Computing and Information Sciences
Radboud University Nijmegen

14 February 2012

Recap

Last week's message

- why analyzing time complexity is important;
- which type of abstractions are done in the analysis
- how to analyze the execution time of a non-recursive algorithm.

Today

- Divide and conquer – (*mergesort* and *quicksort*).
  1. algorithms
  2. solving recurrence relations
- Randomized algorithms – (*randomized quicksort*).

Dive and Conquer

Divide and conquer strategy

- *Divide* the problem into n sub-problems (smaller instances of the original problem)
- *Conquer* the sub-problems
  - trivial for small sizes
  - use the same strategy, otherwise
- *Combine* the solutions of the sub-problems into the solution of the original problem

Implementation is typically recursive (why?)

Merge sort

It uses a *divide and conquer* strategy

- *Divide* the vector in two vectors of similar size
- *Conquer*: recursively order the two vectors (trivial for. . .

Merge sort

It uses a *divide and conquer* strategy

- *Divide* the vector in two vectors of similar size
- *Conquer*: recursively order the two vectors (trivial for. . . vectors of size 1)
- *Combine* two ordered vectors into a new ordered vector. Auxiliary `merge` function.
  1. The `merge` function gets as input two *ordered* sequences `A[p...q]` and `A[q+1...r]`
  2. In the end of the execution we get the sequence `A[p...r]` ordered.

Mergesort

```
void merge_sort(int A[], int p, int r) {
   if (p < r) {                    /* base case ? */
    q = (p+r)/2;                   /* Divide */
    merge_sort(A,p,q);             /* Conquer */
    merge_sort(A,q+1,r);           /* Conquer */
    merge(A,p,q,r);                /* Combine */
    }
}
```

Mergesort

```
void merge_sort(int A[], int p, int r) {
   if (p < r) {                   /* base case ? */
    q = (p+r)/2;                  /* Divide */
    merge_sort(A,p,q);            /* Conquer */
    merge_sort(A,q+1,r);          /* Conquer */
    merge(A,p,q,r);               /* Combine */
    }
}
```

Question: What is the dimension of each sequence in the divide step?

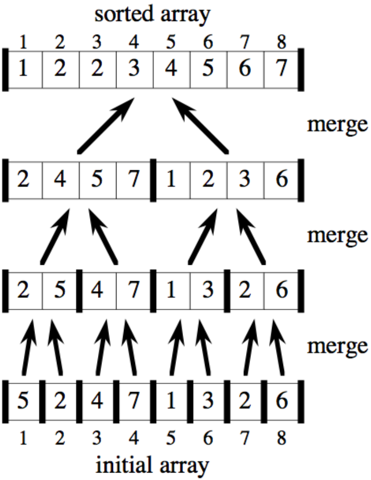
Mergesort

```
void merge_sort(int A[], int p, int r) {
   if (p < r) {                   /* base case ? */
    q = (p+r)/2;                  /* Divide */
    merge_sort(A,p,q);            /* Conquer */
    merge_sort(A,q+1,r);          /* Conquer */
    merge(A,p,q,r);               /* Combine */
    }
}
```

Question: What is the dimension of each sequence in the divide step? $q = \lfloor (p + r)/2 \rfloor$

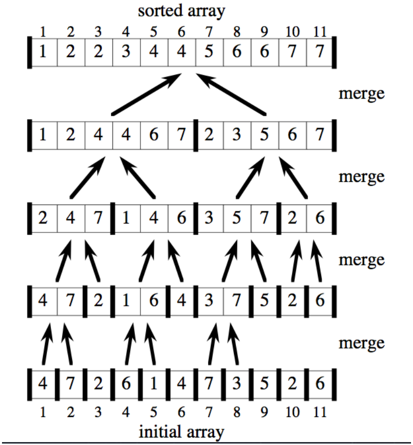
Initial call on an array with `n` elements: `merge_sort(A,1,n)`.

Example I


sorted array
1 2 3 4 5 6 7 8
1 2 2 3 4 5 6 7
merge
2 4 5 7 1 2 3 6
merge
2 5 4 7 1 3 2 6
merge
5 2 4 7 1 3 2 6
1 2 3 4 5 6 7 8
initial array

Example II

Array with number of elements not a multiple of 2



Remains to do...

`merge` procedure

Input: Array A and indices p, q, r such that

- $p \leq q < r$.
- Subarray $A[p..q]$ is sorted and subarray $A[q+1..r]$ is sorted. By the restrictions on p, q, r, neither subarray is empty.

Output: The two subarrays are merged into a single sorted subarray in $A[p..r]$.

Merge

Idea behind merging: Think of two piles of cards.
Each pile is sorted and placed face-up on a table with the smallest cards on top. We will merge these into a single sorted pile, face-down on the table. A basic step:

- Choose the smaller of the two top cards.
- Remove it from its pile, thereby exposing a new top card.
- Place the chosen card face-down onto the output pile.
- Repeatedly perform basic steps until one input pile is empty. Once one input pile empties, just take the remaining input pile and place it face-down onto the output pile.

Merge

Idea behind merging: Think of two piles of cards.
Each pile is sorted and placed face-up on a table with the smallest cards on top. We will merge these into a single sorted pile, face-down on the table. A basic step:

- Choose the smaller of the two top cards.
- Remove it from its pile, thereby exposing a new top card.
- Place the chosen card face-down onto the output pile.
- Repeatedly perform basic steps until one input pile is empty. Once one input pile empties, just take the remaining input pile and place it face-down onto the output pile.

What is the complexity of this algorithm?

Merge

- Each basic step should take constant time, since we check just the two top cards.
- There are $\leq n$ basic steps, since each basic step removes one card from the input piles, and we started with n cards in the input piles.
- Therefore, this procedure should take $\Theta(n)$ time.

The merge procedure

```
void merge(int A[], int p, int q, int r) {
   int L[MAX], R[MAX];
   int n1 = q-p+1;                      /* length of array A[p..q] */
   int n2 = r-q;                        /* length of array A[q+1..r] */
   for (i=1 ; i<=n1 ; i++) L[i] = A[p+i-1];   /* array LEFT */
   for (j=1 ; j<=n2 ; j++) R[j] = A[q+j];     /* array RIGHT */
   L[n1+1] = MAXINT; R[n2+1] = MAXINT;        /* sentinels */
```

The merge procedure

```
void merge(int A[], int p, int q, int r) {
   int L[MAX], R[MAX];
   int n1 = q-p+1;                    /* length of array A[p..q] */
   int n2 = r-q;                      /* length of array A[q+1..r] */
   for (i=1 ; i<=n1 ; i++) L[i] = A[p+i-1];   /* array LEFT */
   for (j=1 ; j<=n2 ; j++) R[j] = A[q+j];     /* array RIGHT */
   L[n1+1] = MAXINT; R[n2+1] = MAXINT;        /* sentinels */
```

what are the sentinels good for?

The merge procedure

```
void merge(int A[], int p, int q, int r) {
   int L[MAX], R[MAX];
   int n1 = q-p+1;                   /* length of array A[p..q] */
   int n2 = r-q;                     /* length of array A[q+1..r] */
   for (i=1 ; i<=n1 ; i++) L[i] = A[p+i-1];   /* array LEFT */
   for (j=1 ; j<=n2 ; j++) R[j] = A[q+j];     /* array RIGHT */
   L[n1+1] = MAXINT; R[n2+1] = MAXINT;        /* sentinels */
   i = 1; j = 1;
   for (k=p ; k<=r ; k++)
     if (L[i] <= R[j]) {     /* put the next smallest
       A[k] = L[i]; i++;       element in the array */
     } else {
       A[k] = R[j]; j++;
     }
}
```

The merge procedure

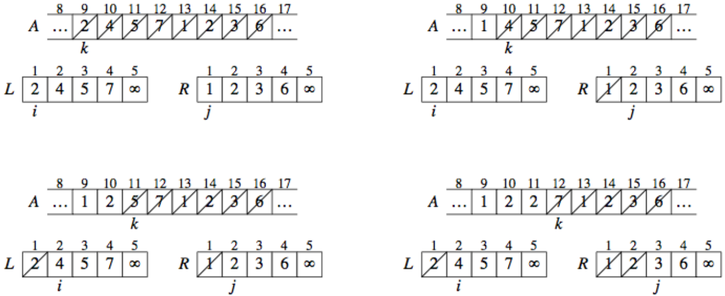
```
void merge(int A[], int p, int q, int r) {
   int L[MAX], R[MAX];
   int n1 = q-p+1;                    /* length of array A[p..q] */
   int n2 = r-q;                      /* length of array A[q+1..r] */
   for (i=1 ; i<=n1 ; i++) L[i] = A[p+i-1];   /* array LEFT */
   for (j=1 ; j<=n2 ; j++) R[j] = A[q+j];     /* array RIGHT */
   L[n1+1] = MAXINT; R[n2+1] = MAXINT;        /* sentinels */
   i = 1; j = 1;
   for (k=p ; k<=r ; k++)
     if (L[i] <= R[j]) {    /* put the next smallest
       A[k] = L[i]; i++;      element in the array */
     } else {
       A[k] = R[j]; j++;
     }
}
```

`merge` executes in $\Theta(n)$ time, where $n = r - p + 1 =$ the number of elements being merged.

Radboud University Nijmegen

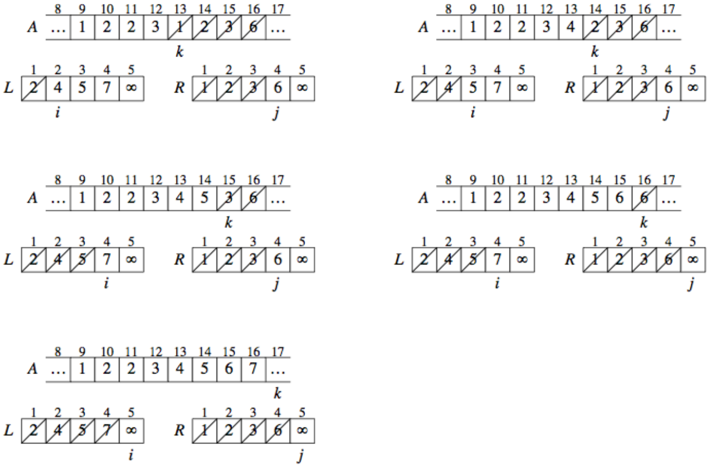
Example

`merge(A,9,12,16)`



Example c'd

`merge(A,9,12,16)`



Complexity analysis

Algorithm with a recursive call: running time can usually be described as a *recurrence*.

Complexity analysis

Algorithm with a recursive call: running time can usually be described as a *recurrence*. How does that work?

Complexity analysis

Algorithm with a recursive call: running time can usually be described as a *recurrence*. How does that work?

Let's for a moment assume that the size of the input is a power of 2. In each *division step* the sub-arrays have exactly size $n/2$.

Complexity analysis

Algorithm with a recursive call: running time can usually be described as a *recurrence*. How does that work?

Let's for a moment assume that the size of the input is a power of 2. In each *division step* the sub-arrays have exactly size $n/2$.

$T(n)$: time of execution (in the worst case) for a input of size n;

If $n = 1$ then $T(n)$ is constant: $T(n) \in \Theta(1)$.

Complexity analysis

Otherwise:

1. **Divide** computing the middle of the vector is done in constant time: $\Theta(1)$.
2. **Conquer** two problems of size $n/2$ are solved: $2T(n/2)$.
3. **Compose** the `merge` function executes in linear time: $\Theta(n)$

Complexity analysis

Otherwise:

1. **Divide** computing the middle of the vector is done in constant time: $\Theta(1)$.
2. **Conquer** two problems of size $n/2$ are solved: $2T(n/2)$.
3. **Compose** the `merge` function executes in linear time: $\Theta(n)$

Hence:

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

Complexity analysis

Otherwise:

1. **Divide** computing the middle of the vector is done in constant time: $\Theta(1)$.
2. **Conquer** two problems of size $n/2$ are solved: $2T(n/2)$.
3. **Compose** the `merge` function executes in linear time: $\Theta(n)$

Hence:

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

and now we want to conclude that $T(n) \in \Theta(??)$.

Merge vs Insertion sort

Running ahead: we will show that $T(n) \in \Theta(n \lg n)$.

Merge vs Insertion sort

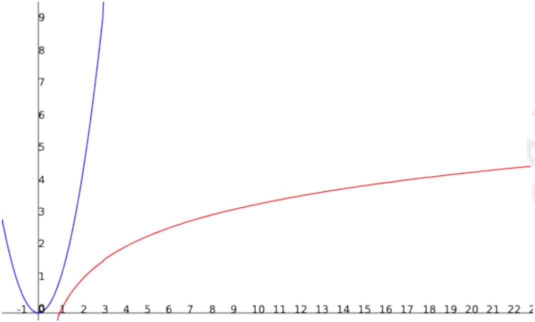
Running ahead: we will show that $T(n) \in \Theta(n \lg n)$.

Cf. Insertion sort in $\Theta(n^2)$.

Merge vs Insertion sort

Running ahead: we will show that $T(n) \in \Theta(n \lg n)$.

Cf. Insertion sort in $\Theta(n^2)$.



Recurrence relations

Going back: How did we get $n \lg n$?

Recurrence relations

Going back: How did we get $n \lg n$?
We had

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

Recurrence relations

Going back: How did we get $n \lg n$?
We had

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

Let us rewrite this as

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

Recurrence relations

Going back: How did we get $n \lg n$?
We had

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

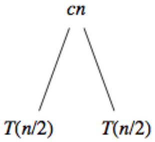
Let us rewrite this as

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

where c is the largest constant representing both the time required to solve problems of dimension 1 as well as the time per array element of the divide and combine steps.

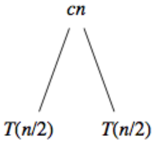
Recurrence relations c'd

For the original problem we have a cost *cn* plus the cost of the sub-problems:

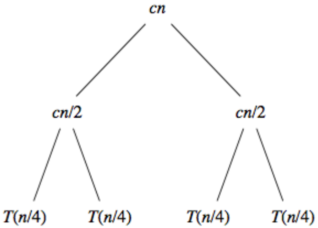


Recurrence relations c'd

For the original problem we have a cost cn plus the cost of the sub-problems:

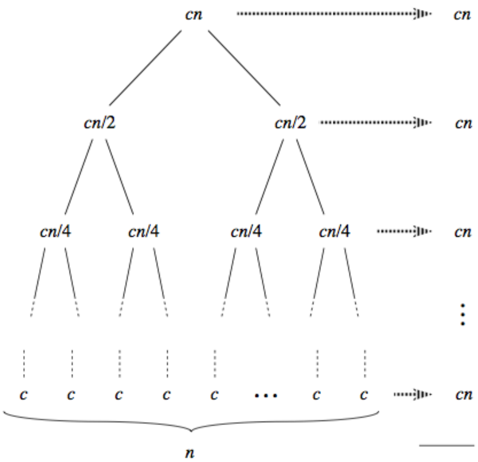


For each of the size-$n/2$ subproblems, we have a cost of $cn/2$, plus two sub-problems, each costing $T(n/4)$:



Recurrence relations c'd

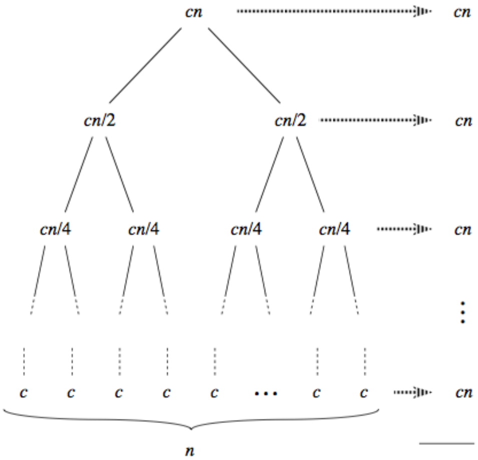
Continue expanding until the problem sizes get down to 1:



- Each level has cost cn.

Recurrence relations c'd

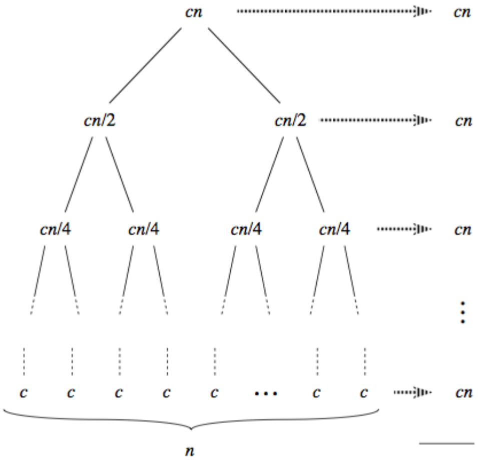
Continue expanding until the problem sizes get down to 1:



- Each level has cost cn.
- There are $\lg n + 1$ levels (height is $\lg n$; proof by induction).

Recurrence relations c'd

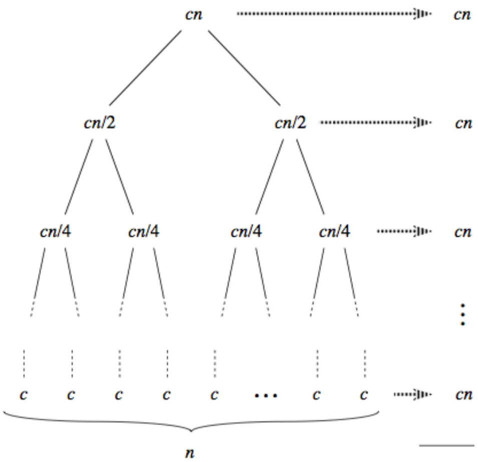
Continue expanding until the problem sizes get down to 1:



- Each level has cost cn.
- There are $\lg n + 1$ levels (height is $\lg n$; proof by induction).
- Total cost is sum of costs at each level: $cn(\lg n + 1) = cn \lg n + cn$.

Recurrence relations c'd

Continue expanding until the problem sizes get down to 1:



- Each level has cost cn.
- There are $\lg n + 1$ levels (height is $\lg n$; proof by induction).
- Total cost is sum of costs at each level: $cn(\lg n + 1) = cn \lg n + cn$.
- Ignore low-order term: `merge sort` worst case is in $\Theta(n \lg n)$.

How to solve recurrences?

First, a small detail: we assumed the length of the array was a power of 2. If that's not case, the right formula for the recurrence would be:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$$

How to solve recurrences?

First, a small detail: we assumed the length of the array was a power of 2. If that's not case, the right formula for the recurrence would be:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$$

The solution of a recurrence can be *verified* using the *substitution method*, which allows us to prove that indeed the recurrence above has solution $T(n) \in \Theta(n \lg n)$

Substituiton method

1. Guess the solution.
2. Use induction to find the constants and show that the solution works.

Example: Consider the following recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

1. Guess: $T(n) = n \lg n + n$ (usually done by looking at the recursion tree).
2. Induction:

Basis: $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$

Substituiton method

Inductive step: Inductive hypothesis is that $T(k) = k \lg k + k$ for all $k < n$.
We willl use this inductive hypothesis for $T(n/2)$.

$$T(n) \quad = \quad 2T(n/2) + n$$

Substituiton method

Inductive step: Inductive hypothesis is that $T(k) = k \lg k + k$ for all $k < n$.
We willl use this inductive hypothesis for $T(n/2)$.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(n/2 \lg n/2 + n/2) + n \quad \text{induction hyp.} \end{aligned}$$

Substituiton method

Inductive step: Inductive hypothesis is that $T(k) = k \lg k + k$ for all $k < n$. We willl use this inductive hypothesis for $T(n/2)$.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(n/2 \lg n/2 + n/2) + n && \text{induction hyp.} \\ &= n \lg n/2 + n + n \end{aligned}$$

Substituiton method

Inductive step: Inductive hypothesis is that $T(k) = k \lg k + k$ for all $k < n$.
We willl use this inductive hypothesis for $T(n/2)$.

$$\begin{array}{rcll} T(n) & = & 2T(n/2) + n & \\ & = & 2(n/2 \lg n/2 + n/2) + n & \text{induction hyp.} \\ & = & n \lg n/2 + n + n & \\ & = & n(\lg n - \lg 2) + n + n & \end{array}$$

Substituiton method

Inductive step: Inductive hypothesis is that $T(k) = k \lg k + k$ for all $k < n$.
We willl use this inductive hypothesis for $T(n/2)$.

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2(n/2 \lg n/2 + n/2) + n \quad \text{induction hyp.} \\
&= n \lg n/2 + n + n \\
&= n(\lg n - \lg 2) + n + n \\
&= n \lg n - n + n + n
\end{aligned}
$$

Substituiton method

Inductive step: Inductive hypothesis is that $T(k) = k \lg k + k$ for all $k < n$.
We willl use this inductive hypothesis for $T(n/2)$.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(n/2 \lg n/2 + n/2) + n \quad \text{induction hyp.} \\ &= n \lg n/2 + n + n \\ &= n(\lg n - \lg 2) + n + n \\ &= n \lg n - n + n + n \\ &= n \lg n + n \end{aligned}$$

Substituiton method

Generally, we use asymptotic notation:

- We would write e.g. $T(n) = 2T(n/2) + \Theta(n)$.
- We assume $T(n) = O(1)$ for sufficiently small n.
- We express the solution by asymptotic notation: $T(n) = \Theta(n \lg n)$ (or $T(n) \in \Theta(n \lg n)$).
- We do not worry about boundary cases, nor do we show base cases in the substitution proof.

Substituiton method

- $T(n)$ is always constant for any constant n.
- Since we are ultimately interested in an asymptotic solution to a recurrence, it will always be possible to choose base cases that work.
- When we want an asymptotic solution to a recurrence, we do not worry about the base cases in our proofs.
- When we want an exact solution, then we have to deal with base cases.

For the substitution method:

- Name the constant in the additive term.
- Show the upper (O) and lower (Ω) bounds separately. Might need to use different constants for each.

Another example

Take the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\lfloor n/2 \rfloor) + n & \text{if } n > 1 \end{cases}$$

We seem to run into a problem if we want to show that $T(n) \leq cn \lg n$:

$$T(1) \leq c1 \lg 1 = 0 \qquad \textit{FALSE}!$$

Another example

Take the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\lfloor n/2 \rfloor) + n & \text{if } n > 1 \end{cases}$$

We seem to run into a problem if we want to show that $T(n) \leq cn \lg n$:

$$T(1) \leq c1 \lg 1 = 0 \qquad \textit{FALSE!}$$

However, recall that the O-notation only requires that we show that there exist $c, n_0 > 0$ for which $T(n) \leq cn \lg n$, for all $n \geq n_0$.

Another example

Hence we just have to replace the base case $n = 1$ by something else... We take it step by step: let's look at $n_0 = 2$

$$T(2) = 4 \leq c2\lg 2 = 2c$$

Enough to choose $c \geq 2$.

Another example

Hence we just have to replace the base case $n = 1$ by something else... We take it step by step: let's look at $n_0 = 2$

$$T(2) = 4 \leq c2\lg 2 = 2c$$

Enough to choose $c \geq 2$.
But now careful:

$$\begin{aligned}
T(2) &= 2T\lfloor 2/2 \rfloor + 2 = 2T(1) + 2 = 4 \\
T(3) &= 2T\lfloor 3/2 \rfloor + 3 = 2T(1) + 3 = 5 \\
T(4) &= 2T\lfloor 4/2 \rfloor + 4 = 2T(2) + 4 = 12 \\
T(5) &= 2T\lfloor 5/2 \rfloor + 5 = 2T(2) + 5 = 13 \\
T(6) &= 2T\lfloor 6/2 \rfloor + 6 = 2T(3) + 6 = 16
\end{aligned}$$

Because both $T(2)$ and $T(3)$ depend on $T(1)$ we cannot simply replace the base case $n = 1$ by $n = 2$, but we need **both** $n = 2$ and $n = 3$.

Another example

Hence we just have to replace the base case $n = 1$ by something else... We take it step by step: let's look at $n_0 = 2$

$$T(2) = 4 \leq c2\lg 2 = 2c$$

Enough to choose $c \geq 2$.
But now careful:

$$\begin{aligned}
T(2) &= 2T\lfloor 2/2 \rfloor + 2 = 2T(1) + 2 = 4 \\
T(3) &= 2T\lfloor 3/2 \rfloor + 3 = 2T(1) + 3 = 5 \\
T(4) &= 2T\lfloor 4/2 \rfloor + 4 = 2T(2) + 4 = 12 \\
T(5) &= 2T\lfloor 5/2 \rfloor + 5 = 2T(2) + 5 = 13 \\
T(6) &= 2T\lfloor 6/2 \rfloor + 6 = 2T(3) + 6 = 16
\end{aligned}$$

Because both $T(2)$ and $T(3)$ depend on $T(1)$ we cannot simply replace the base case $n = 1$ by $n = 2$, but we need **both** $n = 2$ and $n = 3$. For $n = 3$, we have $T(3) = 5 \leq c3\lg 3 = 4.8c$, which also holds for $c \geq 2$.

Master method

Used for many divide-and-conquer recurrences of the form

$$T(n) = aT(n/b) + f(n),$$

where $a \geq 1, b > 1$, and $f(n) > 0$. Based on the *master theorem* (Theorem 4.1 in the book). Compare $n^{\log_b a}$ vs. $f(n)$:

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$. (f (n) is polynomially smaller than $n^{\log_b a}$.)
Solution: $T(n) = \Theta(n^{\log_b a})$.

Case 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$. ($f(n)$ is within a polylog factor of $n^{\log_b a}$, but not smaller.)
Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and $f(n)$ satisfies the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n. ($f(n)$ is polynomially greater than $n^{\log_b a}$.)
Solution: $T(n) = \Theta(f(n))$.

Quicksort

Quicksort is based on the three-step process of divide-and-conquer.

Divide: Partition `A[p..r]`, into two subarrays `A[p..q-1]` and `A[q+1..r]`,such that each element in the first subarray `A[p..q-1]` is $\leq$ than `A[q]` and `A[q]` is $\leq$ each element in the second subarray `A[q+1..r]`.

- The subarrays can be empty;
- computing q is part of the `partition` function
  The `partition` function receives as input `A[p..r]`, partitions it in place using the last element as pivot and returns index q.

Conquer: Sort the two subarrays by recursive calls to `quicksort`.
Combine: No work is needed to combine the subarrays, because they are sorted in place.

Partition

Perform the divide step by a procedure partition, which returns the index q that marks the position separating the subarrays.

```
int partition (int A[], int p, int r) {
x = A[r];
i = p-1;
for (j=p ; j<r ; j++)
 if (A[j] <= x) {
   i++;
   swap(A, i, j);
 }
swap(A, i+1, r);
return i+1;
}
void swap(int X[], int a, int b)
{ aux = X[a]; X[a] = X[b]; X[b] = aux; }
```

Complexity of partition?

Partition

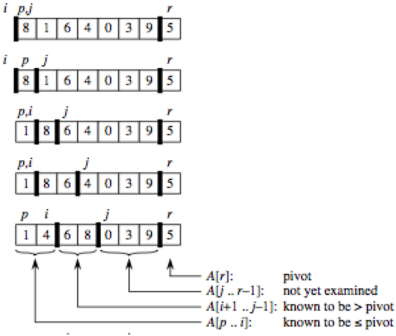
Perform the divide step by a procedure `partition`, which returns the index `q` that marks the position separating the subarrays.

```
int partition (int A[], int p, int r) {
x = A[r];
i = p-1;
for (j=p ; j<r ; j++)
 if (A[j] <= x) {
   i++;
   swap(A, i, j);
 }
swap(A, i+1, r);
return i+1;
}
void swap(int X[], int a, int b)
{ aux = X[a]; X[a] = X[b]; X[b] = aux; }
```

Complexity of partition? linear: $\Theta(n)$.

Example

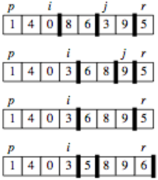
`partition(A,p,r)`



This is the situation after 4 steps in the loop.

Example c'd

`partition(A,p,r)`



Here you see the rest of the steps and final result: the pivot 5 goes in the middle of two sub-arrays that only have, respectively, smaller and larger elements than 5.
(The index *j* disappears because it is no longer needed once the `for` loop is exited.)

Quicksort

```
void quicksort(int A[], int p, int r) {
 if (p < r) {
   q = partition(A,p,r)
   quicksort(A,p,q-1);
   quicksort(A,q+1,r);
 }
}
```

Quicksort

```
void quicksort(int A[], int p, int r) {
 if (p < r) {
   q = partition(A,p,r)
   quicksort(A,p,q-1);
   quicksort(A,q+1,r);
 }
}
```

What is the recurrence relation corresponding to `quicksort`?

Quicksort

```
void quicksort(int A[], int p, int r) {
 if (p < r) {
   q = partition(A,p,r)
   quicksort(A,p,q-1);
   quicksort(A,q+1,r);
 }
}
```

What is the recurrence relation corresponding to `quicksort`?

$$T(n) = D(n) + T(k) + T(k')$$

where $D(n)$ time of partition – $\Theta(n)$ and $k' = n - k - 1$.

Quicksort: analysis

Hence

$$T(n) = \Theta(n) + \max_{k=0}^{n-1}(T(k) + T(n-k-1))$$

Quicksort: analysis

Hence

$$T(n) = \Theta(n) + \max_{k=0}^{n-1}(T(k) + T(n-k-1))$$

Assume $T(n) \leq cn^2$. Then,

Quicksort: analysis

Hence

$$T(n) = \Theta(n) + \max_{k=0}^{n-1}(T(k) + T(n-k-1))$$

Assume $T(n) \leq cn^2$. Then,

$$\begin{aligned} T(n) &\leq \Theta(n) + \max(ck^2 + c(n-k-1)^2) \\ &= \Theta(n) + c\max(k^2 + (n-k-1)^2) \\ &= \Theta(n) + c\max(\underbrace{2k^2 + (2-2n)k + (n-1)^2}_{P(k)}) \end{aligned}$$

When is $P(k)$ maximal?

Quicksort: analysis

Hence

$$T(n) = \Theta(n) + \max_{k=0}^{n-1}(T(k) + T(n-k-1))$$

Assume $T(n) \leq cn^2$. Then,

$$\begin{aligned} T(n) &\leq \Theta(n) + \max(ck^2 + c(n-k-1)^2) \\ &= \Theta(n) + c\max(k^2 + (n-k-1)^2) \\ &= \Theta(n) + c\max(\underbrace{2k^2 + (2-2n)k + (n-1)^2}_{P(k)}) \end{aligned}$$

When is $P(k)$ maximal? $k = 0$ and $k = n-1$: $P(k) = (n-1)^2$.

Quicksort: worst case

Hence, one has

$$T(n) \leq \Theta(n) + c(n-1)^2 = \Theta(n) + c(n^2 - 2n + 1) = \Theta(n) + cn^2$$

which shows that $T(n) \in O(n^2)$.

Quicksort: worst case

Hence, one has

$$T(n) \leq \Theta(n) + c(n-1)^2 = \Theta(n) + c(n^2 - 2n + 1) = \Theta(n) + cn^2$$

which shows that $T(n) \in O(n^2)$.
Is it also true that we have in the worst case $T(n) \in \Theta(n^2)$?

Quicksort: worst case

Hence, one has

$$T(n) \leq \Theta(n) + c(n-1)^2 = \Theta(n) + c(n^2 - 2n + 1) = \Theta(n) + cn^2$$

which shows that $T(n) \in O(n^2)$.
Is it also true that we have in the worst case $T(n) \in \Theta(n^2)$?

- Occurs when **all** the subarrays are completely unbalanced.
- Have **in every recursive call** 0 elements in one subarray and $n-1$ elements in the other subarray. This corresponds to the recurrence

$$T(n) = T(n-1) + \Theta(n) = \sum_{i=0}^{n} \Theta(i) = \Theta(n^2).$$

Quicksort: best case

- In the worst case, `quicksort` is in $\Theta(n^2)$, similar to `insertion sort`, but this worse case occurs when the sequence is already ordered (why?), for which `insertion sort` executes in $\Theta(n)$!

Quicksort: best case

- In the worst case, `quicksort` is in $\Theta(n^2)$, similar to `insertion sort`, but this worse case occurs when the sequence is already ordered (why?), for which `insertion sort` executes in $\Theta(n)$!
- **Best case**: the partition function generates subarrays of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$.

$$T(n) \leq \Theta(n) + 2T(n/2)$$

which has solution $T(n) \in \Theta(n \lg n)$.

Quicksort: best case

- In the worst case, `quicksort` is in $\Theta(n^2)$, similar to `insertion sort`, but this worse case occurs when the sequence is already ordered (why?), for which `insertion sort` executes in $\Theta(n)$!
- **Best case**: the partition function generates subarrays of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$.

$$T(n) \leq \Theta(n) + 2T(n/2)$$

  which has solution $T(n) \in \Theta(n \lg n)$.
- In the **average case**, contrary to most algorithms, `quicksort` is closer to the best case than worst case.

Quicksort: best case

- In the worst case, `quicksort` is in $\Theta(n^2)$, similar to `insertion sort`, but this worse case occurs when the sequence is already ordered (why?), for which `insertion sort` executes in $\Theta(n)$!
- **Best case**: the partition function generates subarrays of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$.

$$T(n) \leq \Theta(n) + 2T(n/2)$$

  which has solution $T(n) \in \Theta(n \lg n)$.
- In the **average case**, contrary to most algorithms, `quicksort` is closer to the best case than worst case.
- Try for instance to check what happens when the subarrays are always divided into $1/9$ and $8/9$ of its size. It will be $\Theta(n \lg n)$.

Quicksort: analysis summary

The running time of quicksort depends on the partitioning of the subarrays:

- If the subarrays are balanced, then `quicksort` can run as fast as `mergesort`.
- If they are unbalanced, then `quicksort` can run as slowly as `insertion sort`.

Some remarks about the average case

- In one execution of quicksort there will be n calls of the `partition` function. (Why?)

Some remarks about the average case

- In one execution of quicksort there will be n calls of the `partition` function. (Why?)
- In general, the execution time is $T(n) = O(n + X)$, where X is the total number of comparisons performed.

Some remarks about the average case

- In one execution of quicksort there will be n calls of the `partition` function. (Why?)
- In general, the execution time is $T(n) = O(n + X)$, where X is the total number of comparisons performed.
- In the average case, we need to analyze, probabilistically, the expected value of *variable* X.

Some remarks about the average case

- In one execution of quicksort there will be n calls of the `partition` function. (Why?)
- In general, the execution time is $T(n) = O(n + X)$, where X is the total number of comparisons performed.
- In the average case, we need to analyze, probabilistically, the expected value of *variable* X.
- Because in reality the `partition` function will not generate in every step subarrays with the same relative size.

Some remarks about the average case

- So we would need to use a *probabilistic distribution* describing the input.
- In the case of a sorting algorithm, we need to know the probability of each permutation of the input occurring.
- When it is not realistic to determine the distribution, we can assume a uniform distribution (all inputs occur with same prob.).
- In such *randomized algorithm* no input corresponds to best or worst case scenario; it is the pre-processing that influences the best and worst case.

Randomized quicksort

One version would be to randomly choose the pivot every time. Hence, the partition function would be modified as follows:

```
int randomized-partition (int A[], int p, int r) {
i = generate_random(p,r) /* random number p<= i <= r */
swap(A,r,i);
return partition(A,p,r);
}
```

We will not dive into details in this course. Expected running time of `quicksort` using randomized-partition is $O(n \lg n)$.

Most important slide of today ;-)

Test

- 1h test, starts at 9 am on Friday
- 3 or 4 questions
- certain: one about O notation
- certain: one about analyzing a non-recursive algorithm
- maybe: one about recurrences