COMP 3711 Design and Analysis of Algorithms

Inversion Number

Divide-and-Conquer

Divide-and-conquer.

- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

Most common pattern.

- Break up problem of size n into two equal parts of size $\frac{1}{2}n$.
- Solve two parts recursively.
- Combine two solutions into overall solution.

Techniques needed.

- Algorithm uses recursion.
- Analysis uses recurrences.

Previous Examples

- Binary Search, Merge Sort

Merge Sort Revision

Merge sort.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

<u>**Mergesort**(A,p,r)**:**</u>

if $p = r$ **then return**

$q \leftarrow \lfloor (p+r)/2 \rfloor$

Mergesort(A,p,q)

Mergesort$(A,q+1,r)$

Merge(A,p,q,r)

<u>**First call:**</u> **Mergesort**$(A,1,n)$

5	2	4	7	1	3	2	6

5	2	4	7	1	3	2	6	divide	$O(1)$
2	4	5	7	1	2	3	6	sort	$2T(n/2)$

1	2	2	3	4	5	6	7	merge	$O(n)$

Inversion Numbers

Def: Given array $A[1..n]$, two elements $A[i]$ and $A[j]$ are inverted if $i < j$ but $A[i] > A[j]$.

- The inversion number of A is the number of inverted pairs.

A useful measure for:

- How "sorted" an array is
- The similarity between two rankings

Songs

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

<u>Inversions</u>

3-2, 4-2

Inversion number = 2

Q: What is the maximum number of inversions if both arrays have size n?

Inversion Number in a Single Array - Relation to Insertion sort

For a single array, the inversion number indicates how har is the array from being sorted, e.g., for sorted array the inversion number is 0

Theorem: The number of swaps used by Insertion Sort = Inversion Number.

Proof: By induction on the size n of the array

Assume Theorem is correct for an array of size $n-1$.

This says that the total number of swaps performed when Insertion Sorting $A[1, n-1]$ is the inversion # of $A[1, n-1]$.

Let $x = A[n]$. Remaining work by the algorithm is

swaps performed when comparing x to items in $A[1, n-1]$

= # of items $j < n$ such that $A[j] > A[n]$,

= # of inversions in which x participates.

Adding these new inversions to the ones in $A[1, n-1]$ gives the full inversion # of $A[1, n]$.

Q: How can we compute the inversion number?

Algorithm 1: Check all $\Theta(n^2)$ pairs.

Algorithm 2: Run Ins sort and count the number of swaps – Also $\Theta(n^2)$ time.

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- Divide: divide array into two halves.
- Conquer: recursively count inversions in each half.
- Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: $\Theta(1)$.

1	5	4	8	10	2

5 blue-blue inversions

6	9	12	11	3	7

8 green-green inversions

Conquer: $2T(n/2)$

9 blue-green inversions
5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Combine: ???

Total = 5 + 8 + 9 = 22.

Counting Inversions: Simple Combine Step

- Assume array is split into left half (blue) and right (green) half and each is already sorted
- How can we count # inversions where a_i and a_j are in different halves?

L[i] →	3	7	10	14	18	19

	2	11	16	17	23	25	← $R[j]$
	6	3	2	2	0	0	← $I[j]$

Count (A,p,q,r) **:**

$L \leftarrow A[p..q], R \leftarrow A[q+1..r]$

L, R already sorted

$i \leftarrow 1, j \leftarrow 1$

$c \leftarrow 0$

While $(i \leq q-p+1)$ && $(j \leq r-q)$

(*) **if** $L[i] \leq R[j]$ **then**

$i \leftarrow i+1$

(**) **else**

$\boldsymbol{I}[\boldsymbol{j}] = q-p-i+2$

$c \leftarrow c + I[j]$

$j \leftarrow j+1$

Let $\boldsymbol{I}[\boldsymbol{j}] =$ # of inversions of $R[j]$ with blue items

Knowing the $\boldsymbol{I}[\boldsymbol{j}]$ solves the problem

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

When $L[i] > R[j]$ and (**) is called,
the blue items $\leq R[j]$ are exactly
the first $i-1$ blue items

The number of blue items greater than $R[j]$, i.e.,
the # of inversions of $R[j]$ with blue items is

$$\boldsymbol{I}[\boldsymbol{j}] = q-p-i+2$$

Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is already (recursively) sorted.
- Count inversions where a_i and a_j are in different halves.
- Merge two sorted halves into sorted whole to maintain sortedness invariant.
- Return # blue-inversions + # green inversions + # blue-green inversions

3	7	10	14	18	19

2	11	16	17	23	25
6	3	2	2	0	0

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

Count: $\Theta(n)$

2	3	7	10	11	14	16	17	18	19	23	25

Merge: $\Theta(n)$

$$T(n) = 2T(n/2) + n, \;\; n > 1$$
$$T(1) = 1$$

So, $T(n) = \Theta(n \log n)$

Counting Inversions: Implementation

Pre-condition. [**Merge-and-Count**] $A[p..q]$ and $A[q+1,r]$ are sorted.
Post-condition. [**Merge-and-Count**] $A[p..r]$ is sorted.

Post-condition. [**Sort-and-Count**] $A[p..r]$ is sorted.

```
Sort-and-Count(A,p,r):
if p = r then return 0
q ← ⌊(p + r)/2⌋
c₁ ← Sort-and-Count(A,p,q)
c₂ ← Sort-and-Count(A,q + 1,r)
c₃ ← Merge-and-Count(A,p,q,r)
return c₁ + c₂ + c₃

First call: Sort-and-Count(A,1,n)
```

```
Merge-and-Count(A,p,q,r):
create two new arrays L and R
L ← A[p..q], R ← A[q + 1..r]
append ∞ at the end of L and R
i ← 1, j ← 1
c ← 0
for k ← p to r
    if L[i] ≤ R[j] then
        A[k] ← L[i]
        i ← i + 1
    else
        A[k] ← R[j]
        j ← j + 1
        c ← c + q − p − i + 2
return c
```

D&C: Observations on Problem Size and Number of Problems

Most common pattern.

- Break up problem of size n into two equal parts of size $\frac{1}{2}n$.
- Solve two parts recursively and combine two solutions into overall solution.

Each time we break up a problem in 2 parts of size $n/2$, we double the number of subproblems and we halve the size of each subroblem

- Level 0, we have the original problem of size n
- Level 1, we break 1 time and we have $2 = 2^1$ problems of size $n/2$
- Level 2, we break 2 times and we have $4 = 2^2$ problems of size $n/2^2$
- Level 3, we break 3 times and we have $8 = 2^3$ problems of size $n/2^3$
- Level i, we break i times and we have 2^i problems of size $n/2^i$

When do we stop breaking up?

- When we cannot break up any more; usually when the problem size becomes 1, i.e., when we reach level i, such that:

$$n/2^i = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n.$$

- The number of problems at (bottom) level logn is: $2^i = 2^{\log_2 n} = n$

D&C: Observations on Problem Size and Number of Problems 2

Other Patterns.

- Break up problem of size n into p parts of size n/q.
- Solve parts recursively and combine solutions into overall solution.

- Level 0, we have the original problem of size n
- Level 1, we break 1 time and we have p problems of size n/q
- Level 2, we break 2 times and we have p^2 problems of size n/q^2
- Level 3, we break 3 times and we have p^3 problems of size n/q^3
- Level i, we break i times and we have p^i problems of size n/q^i

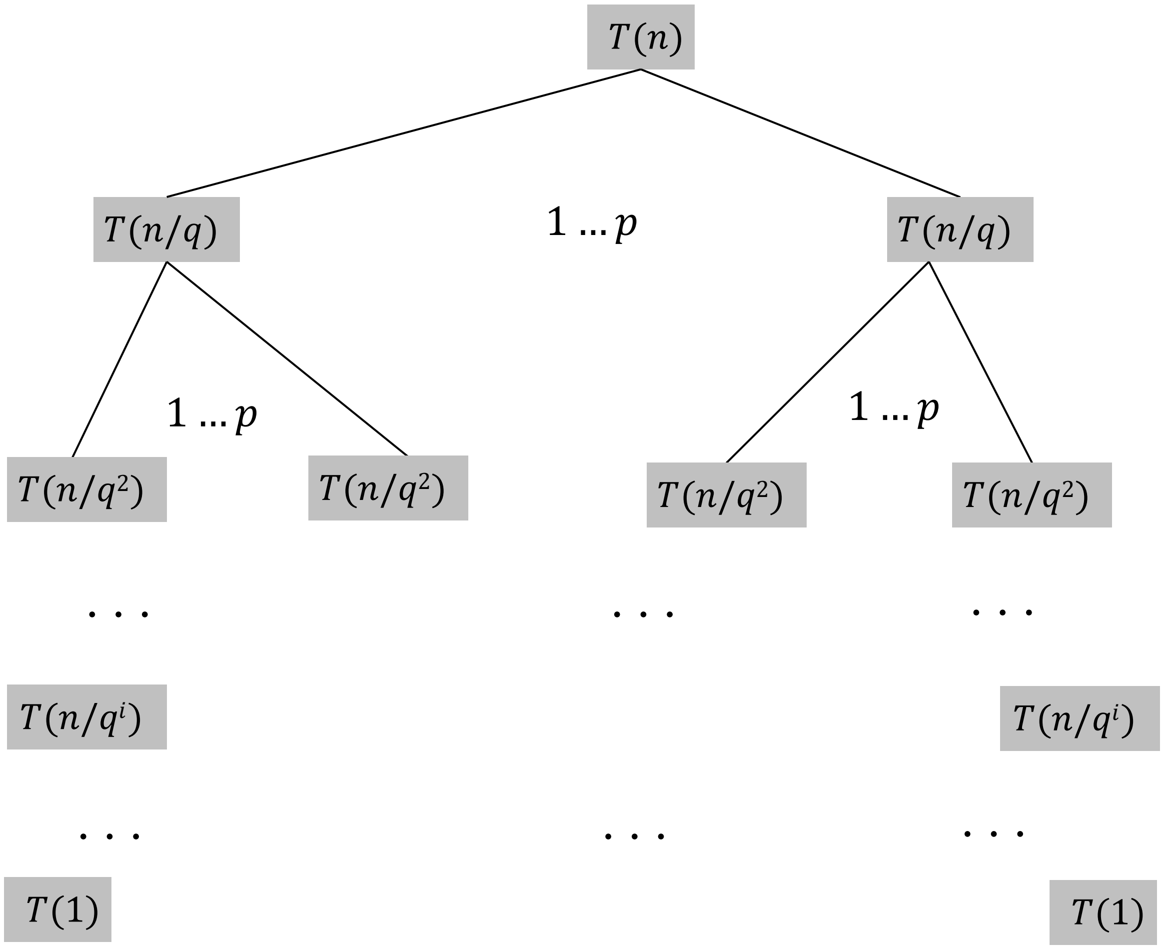
When do we stop breaking up?

- When we cannot break up any more; usually when the problem size becomes 1, i.e., when we reach level i, such that:

  $$n/q^i = 1 \implies n = q^i \implies i = \log_q n.$$

- The number of problems at (bottom) level logn is: $p^i = p^{\log_q n} = n^{\log_q p}$

Visualization



Level	#problems	problem size
0	1	n
1	p	n/q
2	p^2	n/q^2
i	p^i	n/q^i
$\log_q n$	$n^{\log_q p}$	1

- Observation: Assuming T(1)=1, if $p > q \Rightarrow n^{\log_q p}$ > n, which means that the work for the bottom level is superlinear. Therefore the total running time, which includes all levels, cannot be linear.

D&C: Observations on Problem Size and Number of Problems 3

More Patterns.

- Break up problem of size n into $p \geq 2$ parts of size $n - q$.
- Example: for the towers of Hanoi problem, $p = 2$, and $q = 1$: i.e., we break the problem in two problems of size $n - 1$.
- Assume that we break the problem in p problems of size $n - 1$
- Level 1, we break 1 time and we have p problems of size $n - 1$
- Level 2, we break 2 times and we have p^2 problems of size $n - 2$
- Level 3, we break 3 times and we have p^3 problems of size $n - 3$
- Level i, we break i times and we have p^i problems of size $n - i$
- If we stop when the problem size becomes 1, then:

$$n - i = 1 \Longrightarrow i = n - 1.$$

- The number of problems at (bottom) level $n - 1$ is: $p^i = p^{n-1}$
- Assuming T(1)=1, the work p^{n-1} for the bottom level is exponential. Therefore, the total running time, which includes all levels, is also exponential.

Exercise Recursion Tree Method

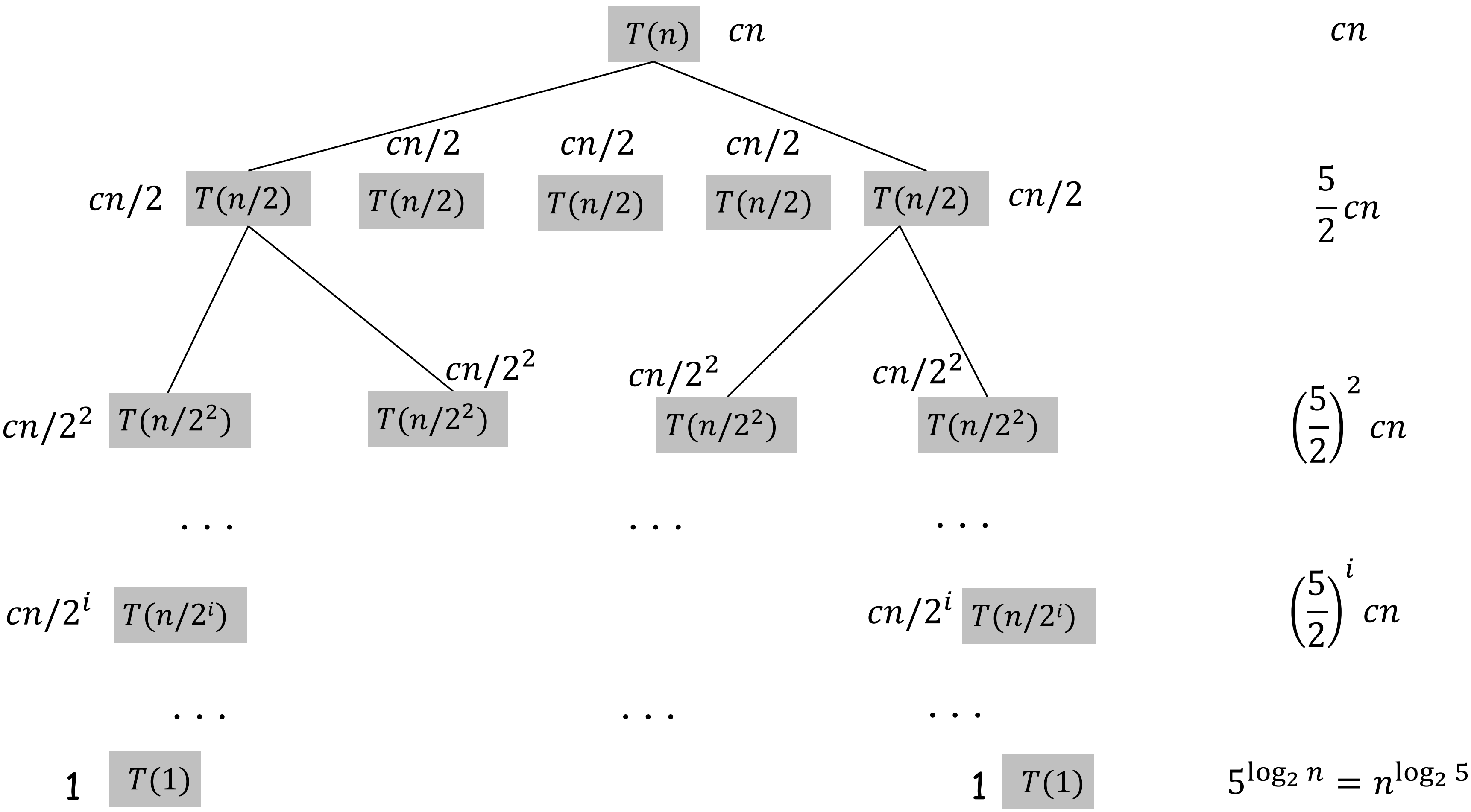
- Algorithm A solves problems of size n by dividing them into 5 subproblems of size $n/2$, recursively solving each subproblem, and then combining the solutions in $O(n)$ time.

- Algorithm B solves problems of size n by recursively solving 2 subproblems of size $n-1$ and then combining the solutions in constant time.

- Algorithm C solves problems of size n by dividing them into 9 subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

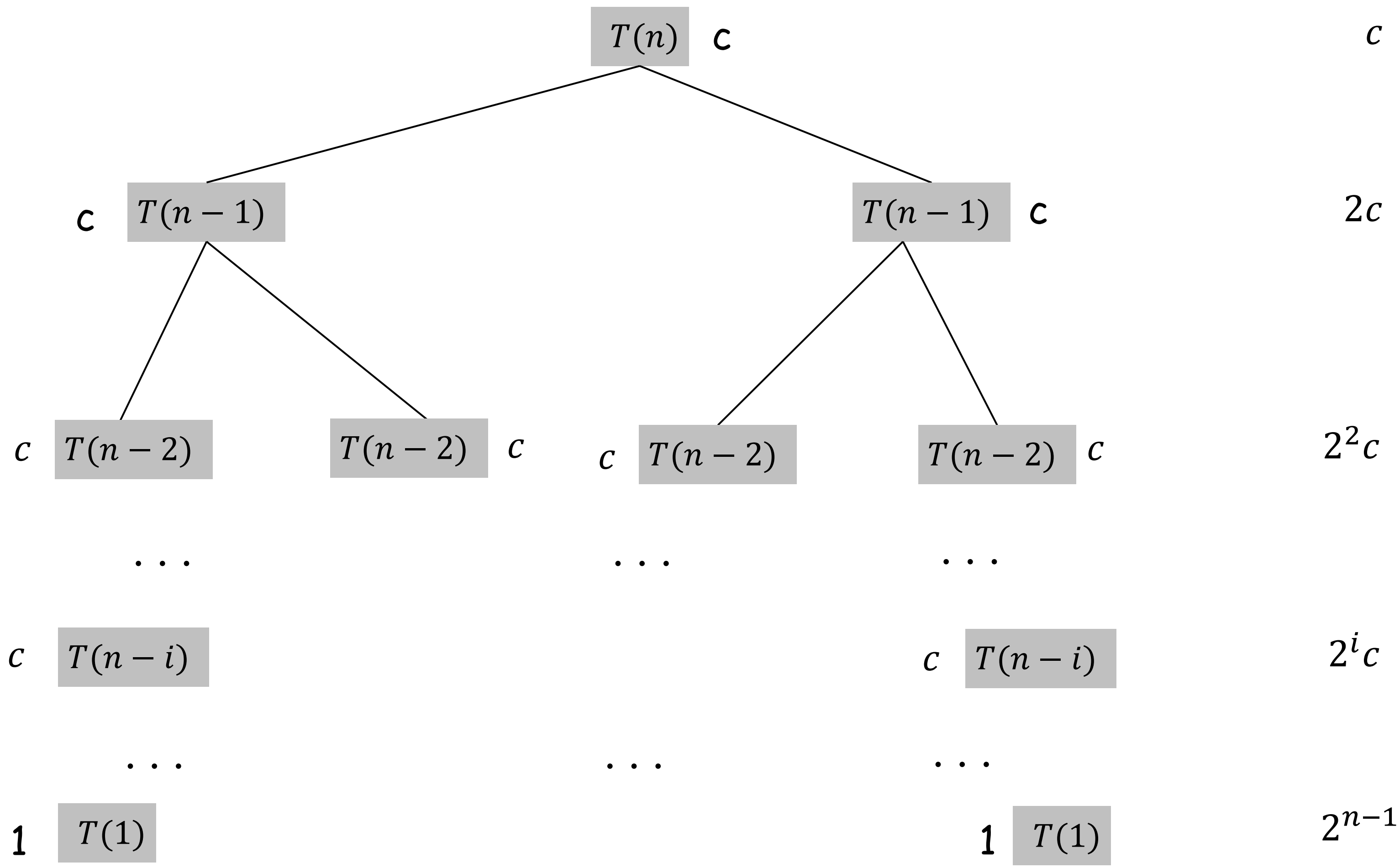
Analyze the running times of these algorithms (in big-O notation) using the recursion tree approach. Determine which is the fastest algorithm asymptotically.

Exercise Algorithm A: $T(n) = 5T\left(\frac{n}{2}\right) + cn$



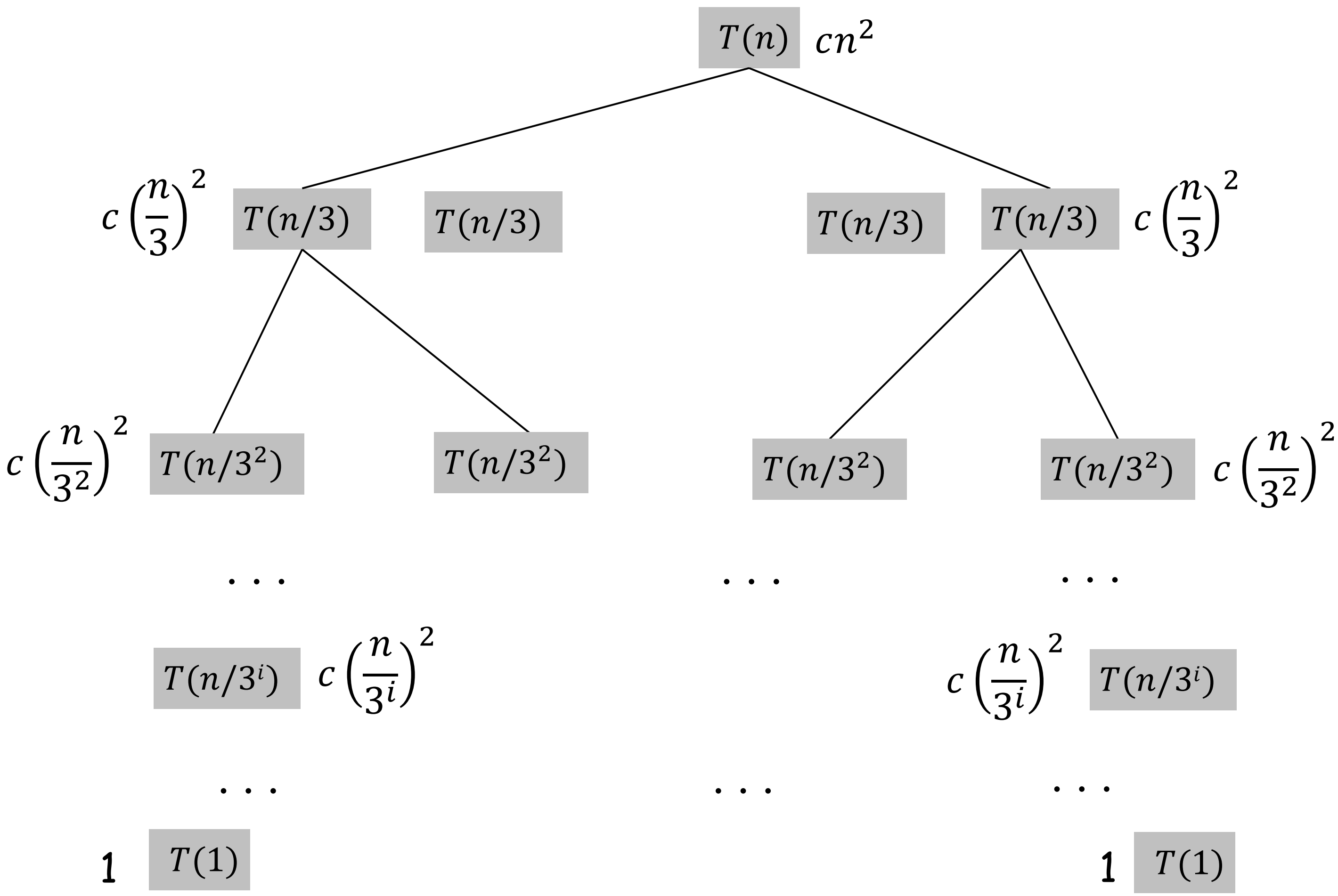
$$T(n) \leq n^{\log_2 5} + cn \sum_{i=0}^{\log_2 n - 1} \left(\frac{5}{2}\right)^i = n^{\log_2 5} + cn \frac{2\left(n^{\log_2 5 - 1} - 1\right)}{3} = O(n^{\log_2 5})$$

Exercise Algorithm B: $T(n) = 2T(n-1) + c$



$$T(n) \le 2^{n-1} + c \sum_{i=0}^{n-2} 2^i = O(2^n)$$

Exercise Algorithm C: $T(n) = 9T\left(\frac{n}{3}\right) + cn^2$



cn^2

$9 \times c\left(\frac{n}{3}\right)^2 = cn^2$

$9^2 \times c\left(\frac{n}{3^2}\right)^2 = cn^2$

$9^i \times c\left(\frac{n}{3^i}\right)^2 = cn^2$

$9^{\log_3 n} = n^{\log_3 9} = n^2$

$$T(n) \le n^2 + \sum_{i=0}^{\log_3 n - 1} cn^2 = O(n^2 \log n)$$

Exercise D&C for finding the max and the min in an array

Assume, for simplicity that n is a power of 2: $n = 2^{k+1}$.

Idea: if $n = 2$ $(k = 0)$, with one comparison you can return the max and the min

If $n > 2$ $(k > 0)$, split the array in two parts L and R. Recursively find maxL, minL, and maxR, minR. Then, return max(maxL, maxR) and min(minL, minR).

```
DQ-find-max-min(A[p..r])
if r = p + 1 then
        if A[p] > A[r]  RETURN(A[p], A[r])
        else RETURN(A[r], A[p])
 mid ← ⌊(p + r)/2⌋
(maxL,minL)=DQ-find-max-min(A[p..mid])
(maxR,minR)=DQ-find-max-min(A[mid + 1..r])
if   maxL>maxR then maxA ← maxL
                else maxA ← maxR
if   minL<minR then minA ← minL
                else minA ← minR
RETURN(maxA, minA)
```

Exercise (cont) D&C for finding the max and the min in an array - Number of comparisons

If $n = 2$ $(k = 0)$, with one comparison, I can return the max and the min. $T(2) = 1$.

If $n > 2$ $(k > 0)$, 2 comparisons to return max(maxL, maxR) and min(minL, minR).

Recurrence: $T(n) = 2T(n/2) + 2$. Solving the recurrence:

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

$= 2^3T(n/2^3) + 2^3 + 2^3 + 2$

...

$= 2^iT\left(n/2^i\right) + 2^i + 2^{i-1} + \cdots + 2$

for step i.

We continue, like that until the boundary case where the problem size equals 2, i.e., when we reach step k. Recall that $n = 2^{k+1}$. For this step we have: $2^kT\left(n/2^k\right) + 2^k + 2^{k-1} + \cdots + 2 = (n/2)T(2) + 2^k + 2^{k-1} + \cdots + 2$.

$(n/2)T(2) = n/2$.

$2^k + 2^{k-1} + \cdots + 2 = 2\left(2^k - 1\right) = 2(n/2 - 1) = n - 2$

Total number of comparisons: $n/2 + n - 2 = 3n/2 - 2$.