COMP 251

Algorithms & Data Structures (Winter 2022)

Algorithm Paradigms – Divide and Conquer

School of Computer Science
McGill University

Slides of (Comp321 ,2021), Langer (2014), slides by K. Wayne & Snoeyink and (Kleinberg & Tardos, 2005)

Announcements

Outline

- Complete Search
- Divide and Conquer.
  - Introduction.
  - Examples.
- Dynamic Programming.
- Greedy.

Algorithmic Paradigms – Divide and Conquer

- It is a problem solving paradigm where we try to make a problem simpler by 'dividing' it into smaller parts and 'conquering' them.

smaller = simpler

- Recursive in structure

① - ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
    - Usually by half or nearly half.

② - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.

③ - ***Combine*** the solutions to create a solution to the original problem

Decrease and Conquer

- Sometimes we're not actually dividing the problem into many subproblems, but only into one smaller subproblem
- Usually called decrease and conquer
- The most common example of this is binary search O(log n).

  - Given a sorted array of elements.
    1. Base case: the array is empty, return false
    2. Compare x to the element in the middle of the array
    3. If it's equal, then we found x and we return true
    4. If it's less, then x must be in the left half of the array
       4.1 Binary search the element (recursively) in the left half
    5. If it's greater, then x must be in the right half of the array
       5.1 Binary search the element (recursively) in the right half

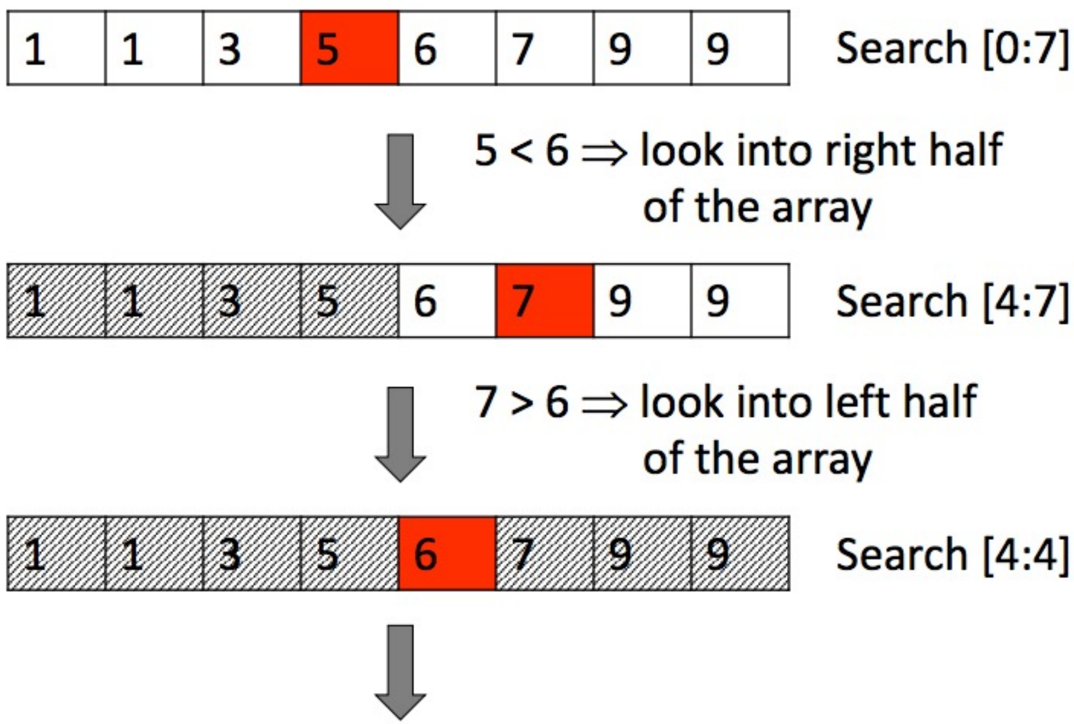
Decrease and Conquer

Example: Does the following **sorted** array A contains the number 6?

A =

1	1	3	5	6	7	9	9

Call: `binarySearch(A, 0, 7, 6)`



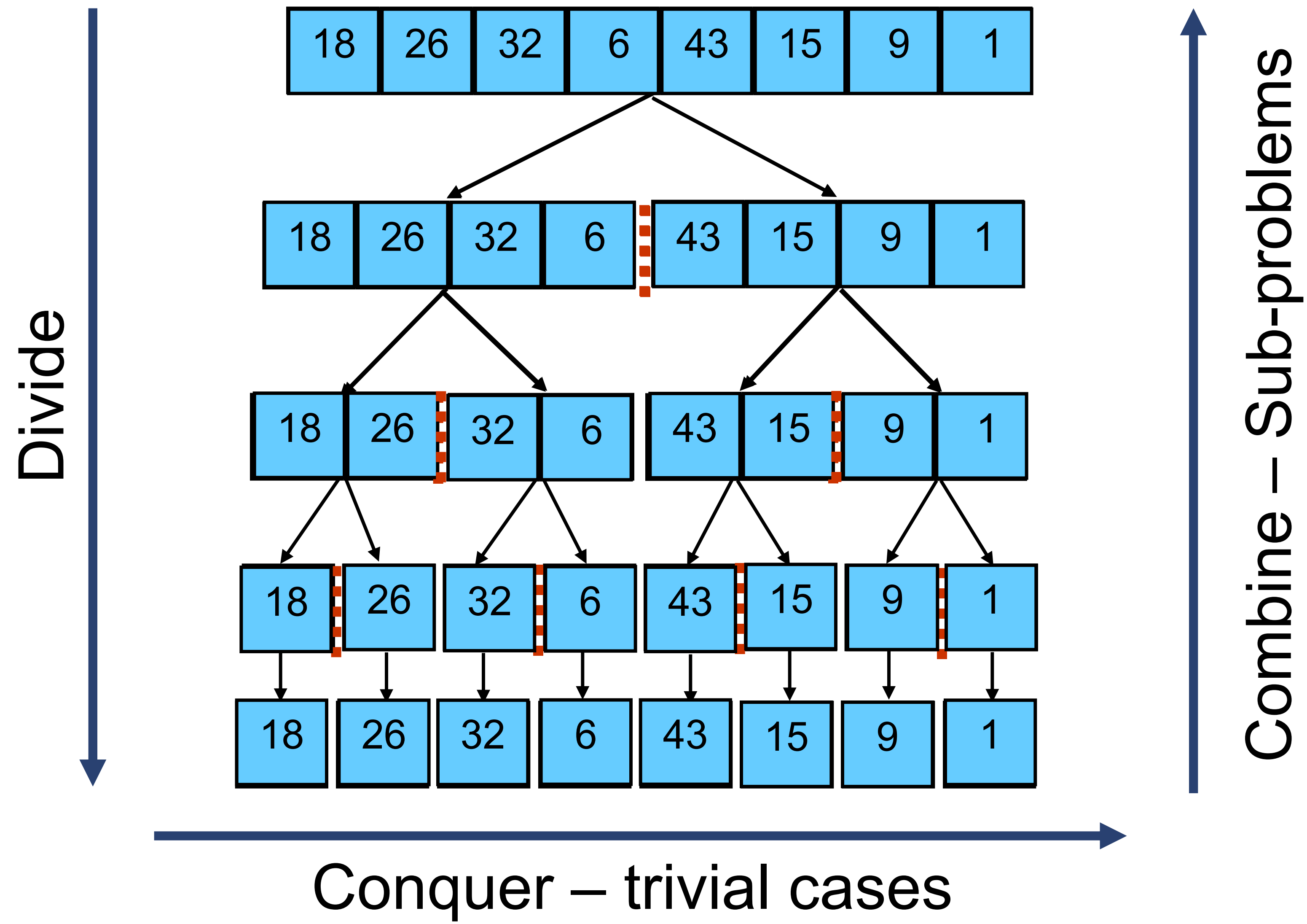
6 is found. Return 4 (index)

Divide and Conquer – Merge Sort

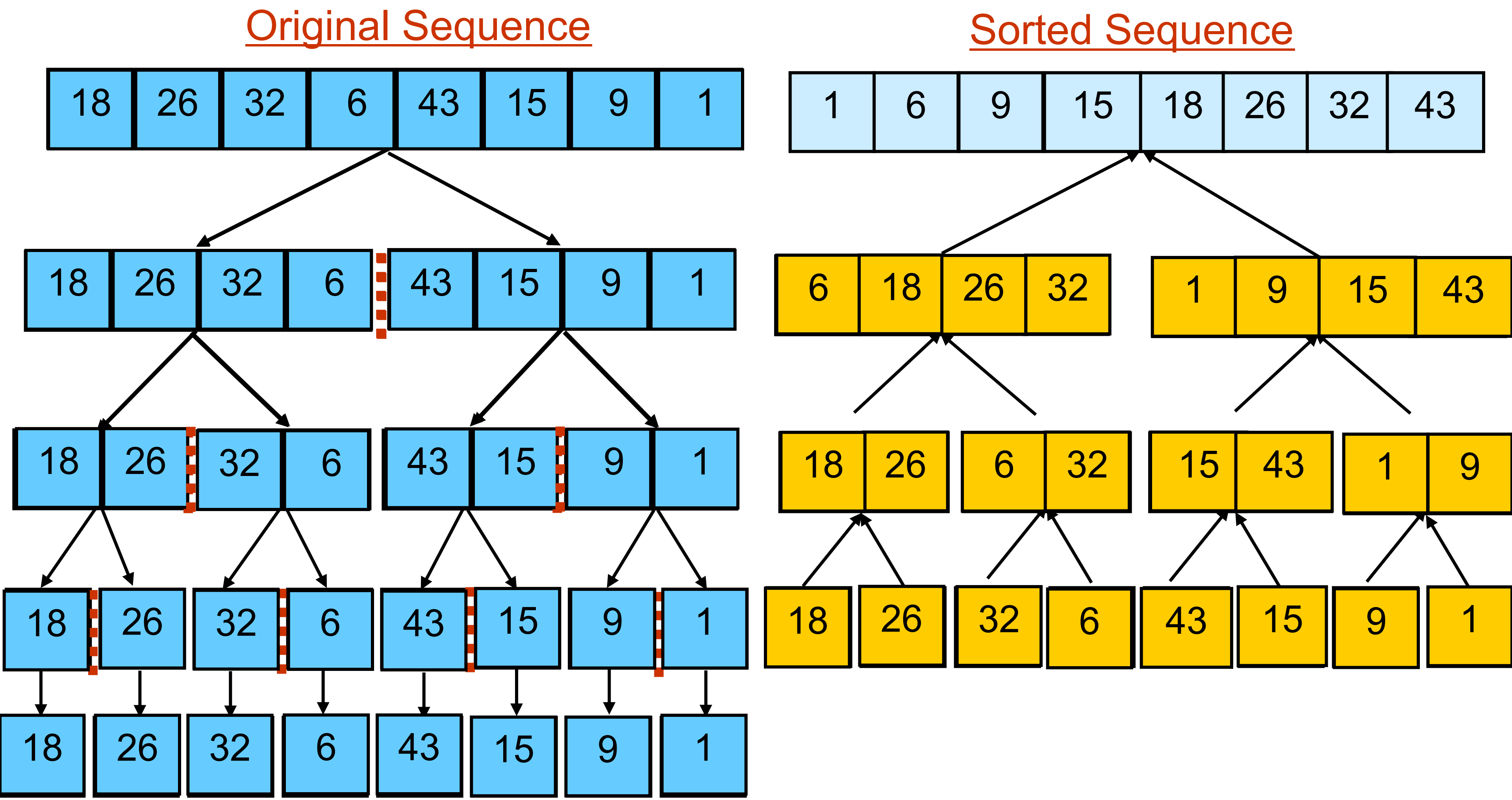
***Sorting Problem*:** Sort a sequence of *n* elements into non-decreasing order.

- ***Divide***: Divide the *n*-element sequence to be sorted into two subsequences of *n/2* elements each
- ***Conquer:*** Sort the two subsequences recursively using merge sort.
- ***Combine***: Merge the two sorted subsequences to produce the sorted answer.

Divide and Conquer – Merge Sort



Divide and Conquer – Merge Sort



MergeSort(A, p, r)

INPUT: **a sequence of *n* numbers stored in array A**

OUTPUT: **an ordered sequence of *n* numbers**

```
MergeSort (A, p, r)   // sort A[p..r] by divide & conquer
  if p < r
     then q ← ⌊(p+r)/2⌋
          MergeSort (A, p, q)
          MergeSort (A, q+1, r)
          Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]
```

Initial Call: MergeSort(A, 1, n)

Merge(A, p, q, r)

Merge(*A*, *p*, *q*, *r*)

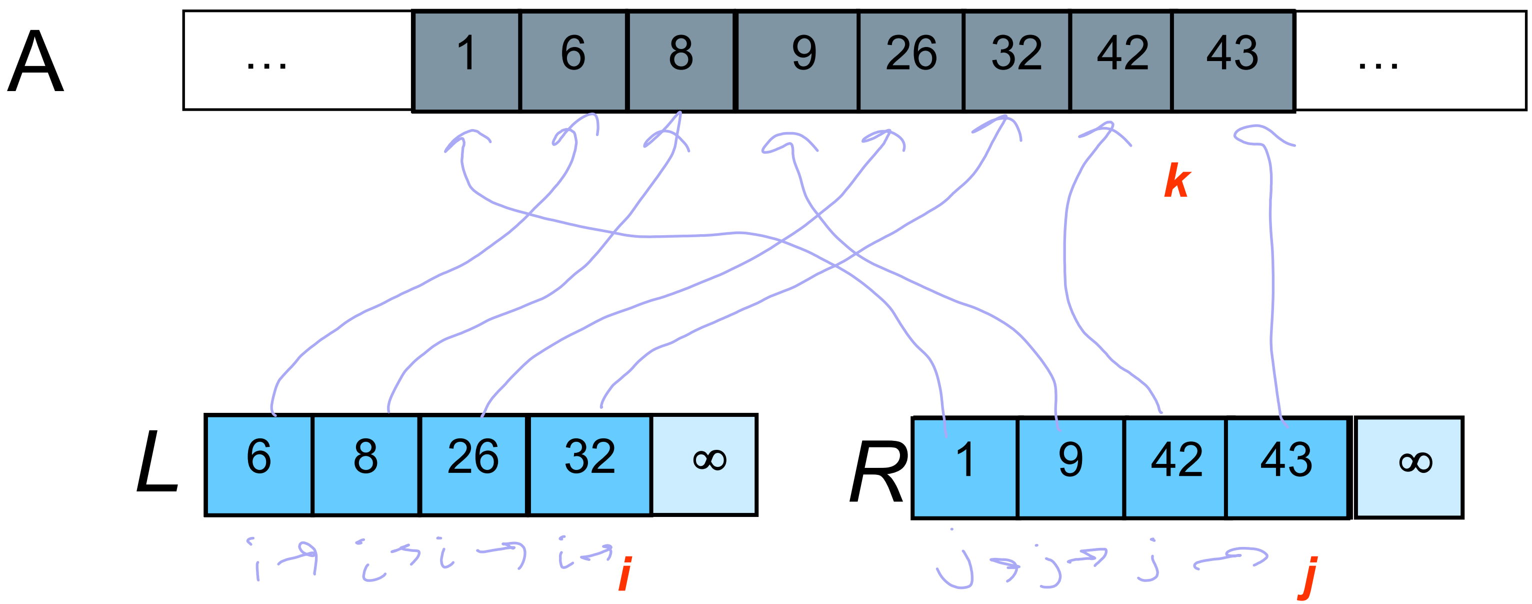
```
n1 ← q – p + 1
n2 ← r – q
for i ← 1 to n1
    do L[i] ← A[p + i – 1]
for j ← 1 to n2
    do R[j] ← A[q + j]
L[n1+1] ← ∞
R[n2+1] ← ∞
i ← 1
j ← 1
for k ←p to r
    do if L[i] ≤ R[j]
        then A[k] ← L[i]
             i ← i + 1
        else A[k] ← R[j]
             j ← j + 1
```

Input: Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.

Output: Merged sorted subarray in $A[p..r]$.

Sentinels, to avoid having to check if either subarray is fully copied at each step.

Merge - Example



Merge - Correctness

Merge(A, p, q, r)

```
n1 ← q – p + 1
n2 ← r – q
for i ← 1 to n1
    do L[i] ← A[p + i – 1]
for j ← 1 to n2
    do R[j] ← A[q + j]
L[n1+1] ← ∞
R[n2+1] ← ∞
i ← 1
j ← 1
for k ← p to r
    do if L[i] ≤ R[j]
        then A[k] ← L[i]
                i ← i + 1
        else A[k] ← R[j]
                j ← j + 1
```

Loop Invariant property (main *for* loop)

- At the start of each iteration of the `for` loop, subarray $A[p..k-1]$ contains the $k-p$ smallest elements of L and R in sorted order.
- $L[i]$ and $R[j]$ are the smallest elements of L and R that have not been copied back into A.

Initialization:

Before the first iteration:

- $A[p..k-1]$ is empty, k = p => k – p = 0.
- $i = j = 1$.
- $L[1]$ and $R[1]$ are the smallest elements of L and R not copied to A.

Merge - Correctness

Merge(*A*, *p*, *q*, *r*)

```
1.    n1 ← q – p + 1
2.    n2 ← r – q
3.    for i ← 1 to n1
4.        do L[i] ← A[p + i – 1]
5.    for j ← 1 to n2
6.        do R[j] ← A[q + j]
7.    L[n1+1] ← ∞
8.    R[n2+1] ← ∞
9.    i ← 1
10.   j ← 1
11.   for k ← p to r
12.       do if L[i] ≤ R[j]
13. case1     then A[k] ← L[i]
14.                i ← i + 1
15.            else A[k] ← R[j]
16. case 2         j ← j + 1
```

Maintenance:

Case 1: $L[i] \le R[j]$

- By LI, *A* contains $k - p$ smallest elements of *L* and *R* in sorted order.
- By LI, $L[i]$ and $R[j]$ are the smallest elements of *L* and *R* not yet copied into *A*.
- Line 13 results in *A* containing $k - p + 1$ smallest elements (again in sorted order). Incrementing *i* and *k* reestablishes the LI for the next iteration.

Case 2: Similar arguments with $L[i] > R[j]$

Termination:

- On termination, $k = r + 1$. By LI, *A[p..k-1], which is A[p..r],* contains $k\text{-}p=r–p+1$ smallest elements of *L* and *R* in sorted order.
- *L* and *R* together *contain* $n_1 + n_2 + 2 = r – p + 3$ elements including the two sentinels. All but the two largest (i.e., sentinels) have been copied in A.

MergeSort - Analysis

- Running time ***T*(*n*)** of Merge Sort:
  - falls out from the three steps of the basic paradigm
- Divide: computing the middle takes O(1)
- Conquer: solving 2 subproblems takes $2T(n/2)$
- Combine: merging n elements takes $O(n)$
- Total:

$$T(n) = O(1) \quad \text{if } n = 1$$
$$\boxed{T(n) = 2T(n/2) + O(n) + O(1)} \quad \boxed{\text{if } n > 1}$$

$$\Rightarrow \boxed{T(n) = O(n \lg n)}$$

In general – Analysis - Recurrence

Trivial case of size <= constant

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\,, \\ \underbrace{a}_{\text{\# of subproblems}}\,T(\underbrace{n/b}_{\text{size of subproblems}}) + \underbrace{D(n)}_{\text{divide time}} + \underbrace{C(n)}_{\text{combine time}} & \text{otherwise}\,. \end{cases}$$

Solving recurrences

- ***Substitution method***: we guess a bound and then use mathematical induction to prove that our guess is correct.
- ***Recursion-tree method:*** converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
- ***Master method:*** provides bounds for recurrences of the form.

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

where $a \geq 1$, $b > 1$, and $f(n)$ is a given function

MergeSort – Substitution method

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

assuming n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2\,T(n/2) + n & \text{otherwise} \end{cases}$$

Pf 2. [by induction on n]

- Base case: when $n = 1$, $T(1) = 0$.
- Inductive hypothesis: assume $T(n) = n \log_2 n$.
- Goal: show that $T(2n) = 2n \log_2 (2n)$.

because we have even time

$$\log_2 2n = \log_2 2 + \log_2 n$$
$$\log_2 2n - 1 = \log_2 n$$

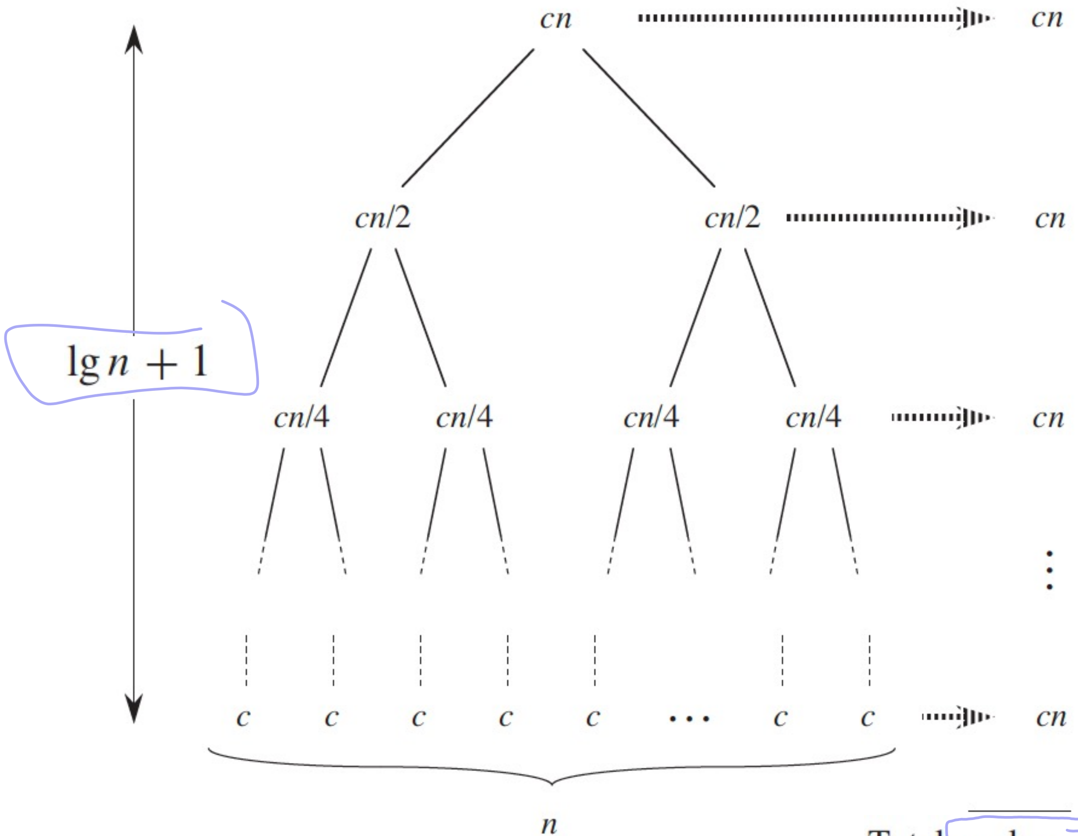
$$\begin{aligned} T(2n) &= 2\,T(n) + 2n \\ &= 2n \log_2 n + 2n \\ &= 2n (\log_2 (2n) - 1) + 2n \\ &= 2n \log_2 (2n). \quad \blacksquare \end{aligned}$$

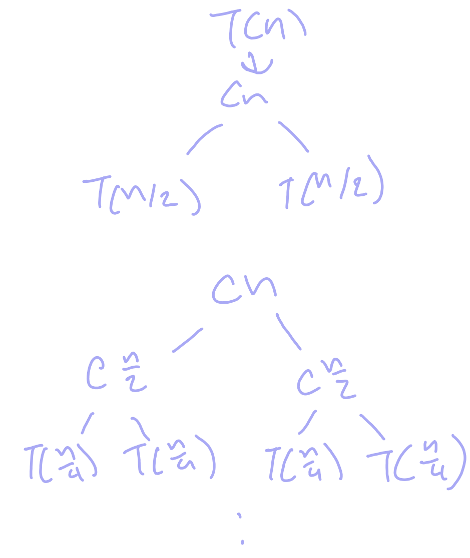
2n instead of n+1

MergeSort – Recursion Tree

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

assuming n is a power of 2





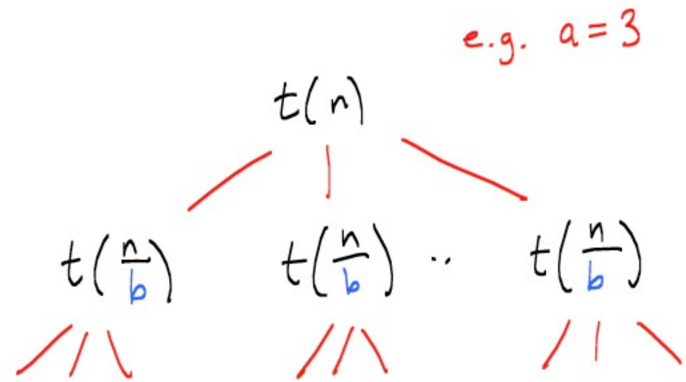
Total: $\boxed{cn \lg n + cn}$ → O(n lg n)

Recursion Tree

Suppose we have a divide and conquer algorithm that gives a recurrence:

$$\boxed{t(n) = a\, t\left(\frac{n}{b}\right) + c\, n^d}$$

Notice that a, b and d are independent of n



- a is the number of subproblems
- $\frac{n}{b}$ is the size of each subproblem
- n^d is the overhead for problem of size n (to partition and combine solutions)

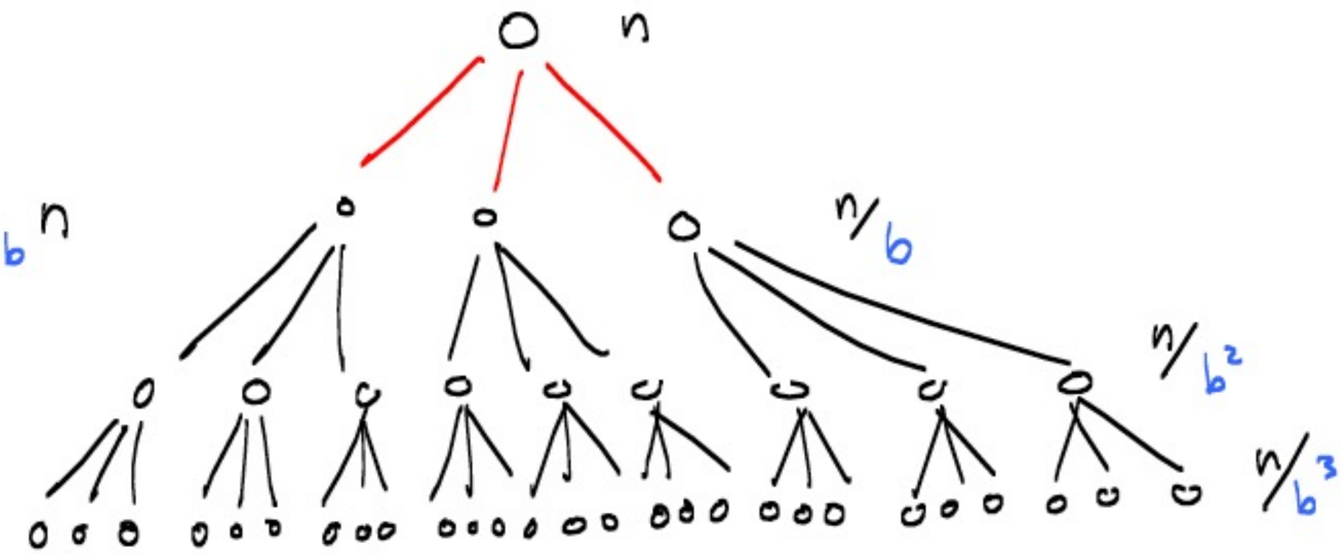
Note that n^d represents the work done outside the recursive call

I'll set $c = 1$.

Recursion Tree

$$t(n) = a \; t\left(\frac{n}{b}\right) + c\, n^d$$

height of call tree $= \log_b n$



number of leaves. $= a^{\log_b n}$

Recursion stops at the base case, typically when problem size is a small number

Recursion Tree – Good VS Evil

$$\boxed{t(n) = a\, t\left(\frac{n}{b}\right) + c\, n^d}$$

assume $c=1$

Tim Roughgarden :

" the forces of good and evil "

good - the size of each subproblem shrinks with each recursive call $(b>1)$

evil - the number of subproblems increases at each level of the call tree. $(a>1)$

Let’s the battle begin

Taken from pinterest

Let's the battle begin (supplemental)

- But first lets define (recall) the allowed 'super powers'.
- Geometric series power (convergence power).
  - Sum of a number of terms that have a constant ratio between successive terms.

$$\frac{1}{1}+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\ldots\ldots\ldots \rightarrow r=\frac{1}{2}$$

  - In general:

$$S_n = \sum_{k=0}^{n} r^k = 1 + r + r^2 + \cdots + r^n$$

  - Multiplying both sides by r gives.

$$\mathrm{r}S_n = r + r^2 + r^3 + \cdots + r^{n+1}$$

Let's the battle begin (supplemental)

- Geometric series (convergence power).
  - Substracting the two previous equations.

$$(1-r)S_n = (1+\cancel{r}+\cancel{r^2}+\cdots+\cancel{r^n}) - (\cancel{r}+\cancel{r^2}+\cancel{r^3}+\cdots+r^{n+1})$$

$$(1-r)S_n = 1 - r^{n+1}$$

$$\boxed{S_n = \sum_{k=0}^{n} r^k = \frac{1-r^{n+1}}{1-r}}$$

  - For -1 < r < 1, the sum converges as $n \to \infty$, in which case

$$\boxed{S_\infty = \sum_{k=0}^{\infty} r^k = \frac{1}{1-r}}$$

Let's the battle begin (supplemental)

- Exponents and logs (manipulation power).

$$x^{(yz)} = (x^y)^z \neq x^{(y^z)}$$

e.g. $$x^{2\cdot 3} = (x^2)^3 \neq x^{(2^3)}$$

$$= (x^2)(x^2)(x^2) \qquad = x^8$$

$$= x^6$$

e.g $$(b^d)^{\log_b n} = b^{d \log_b n}$$

$$= (b^{\log_b n})^d$$

$$= n^d$$

Let’s the battle begin (supplemental)

- Exponents and logs (manipulation power).

for any $a, b, x > 0$

$$\log_b x = \log_b a \cdot \log_a x$$

Why?

$$x \equiv a^{\log_a x}$$

take $\log_b$ of both sides

$$\log_b x = \log_b \left(a^{\log_a x} \right)$$

$$= \log_a x \cdot \log_b a$$

Let's the battle begin (supplemental)

- Exponents and logs (manipulation power).

<u>Claim</u>: $a^{\log_b n} = n^{\log_b a}$

<u>Proof</u>:

$$
\begin{aligned}
a^{\log_b n} &= a^{\log_b a \, \log_a n} \\
&= \left(a^{\log_a n} \right)^{\log_b a} \\
&= n^{\log_b a}
\end{aligned}
$$

Good VS Bad – battle

Assume $n = b^k$ for simplicity

$$t(n) = \overbrace{a}^{a \text{ subproblems}} \, t\Big(\underbrace{\frac{n}{b}}_{\text{subproblem size}}\Big) + \overbrace{n^d}^{\text{cost}}$$

level 2

$$= a\left[a\,t\left(\frac{n}{b^2}\right) + \left(\frac{n}{b}\right)^d\right] + n^d$$

$$= a^2\,t\left(\frac{n}{b^2}\right) + a\left(\frac{n}{b}\right)^d + n^d$$

$$a\,t\left(\frac{n}{b^3}\right) + \left(\frac{n}{b^2}\right)^d$$

level 3

$$= a^{\overset{\text{level}}{3}}\,t\left(\frac{n}{b^3}\right) + a^2\left(\frac{n}{b^2}\right)^d + a\left(\frac{n}{b}\right)^d + n^d$$

$\vdots$ keep going

Good VS Bad – battle

$n = b^k$

level k

$$= a^k\, t\left(\frac{n}{b^k}\right) + \sum_{i=0}^{k-1} a^i \left(\frac{n}{b^i}\right)^d$$

last level $\log_b n$

$$= \underbrace{a^{\log_b n}}_{\text{number of leaves}}\ \underbrace{t(1)}_{\text{work at each leaf (base case)}} + \sum_{i=0}^{\log_b n - 1} \underbrace{a^i}_{\text{number of internal nodes at level } i} \underbrace{\left(\frac{n}{b^i}\right)^d}_{\text{work at each internal node at level } i}$$

Good VS Bad – battle

insert

$$t(n) = a^{\log_b n}\, t(1) + \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^d$$

$$= n^d \sum_{i=0}^{\log_b n} \underbrace{\left(\frac{a}{b^d}\right)}^{i}$$

a = bad guy
(# subproblems)

b = good guy
(shrinking problem)

Note that the battle is this ratio.

Assume $t(1) = 1$, and note $\frac{n}{b^{\log_b n}} = 1$.

Good VS Bad – battle possible results

$$t(n) = n^d \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i$$

- If $a < b^d$ (good wins)
  - The amount of work is decreasing with the recursion level i.
  - Worst case is in the root (i.e., $i = 0$)
    - Might expect $O(n^d)$ => The work n^d of the root dominates
- If $a = b^d$ (there is a tie)
  - The amount of work is the same at every recursion level i.
  - All levels have the same 'worst' case.
    - Might expect $O(n^d \log n)$ => n^d for all the *log* levels
- If $a > b^d$ (evil wins)
  - The amount of work is increasing with the recursion level i.
  - Worst case is in the leaves (i.e., $i = \log_b n$)
    - Might expect => $O(n^{\log(a)})$ => O(#leaves) because leaves dominates

Good VS Bad – battle possible results

$$t(n) = n^{d} \sum_{i=0}^{\log_b n} \left(\frac{a}{b^{d}}\right)^{i}$$

geometric series

$$\sum_{i=0}^{k} r^{i}$$

three cases

1) $r = 1$
2) $r < 1$
3) $r > 1$

If $a = b^d$ (there is a tie)
If $a < b^d$ (good wins)
If $a > b^d$ (evil wins)

Case 1 ($r = 1$) : $a = b^d$

Here we have the same amount of work at each level.

$$1 + r + r^2 + r^3 + \dots r^k$$

$r = 1$

$$= 1 + 1 + 1 + 1 + \dots 1$$

$$= k + 1$$

$$= \log_b n + 1$$

$$\Rightarrow \quad t(n) = O(n^d \log_b n)$$

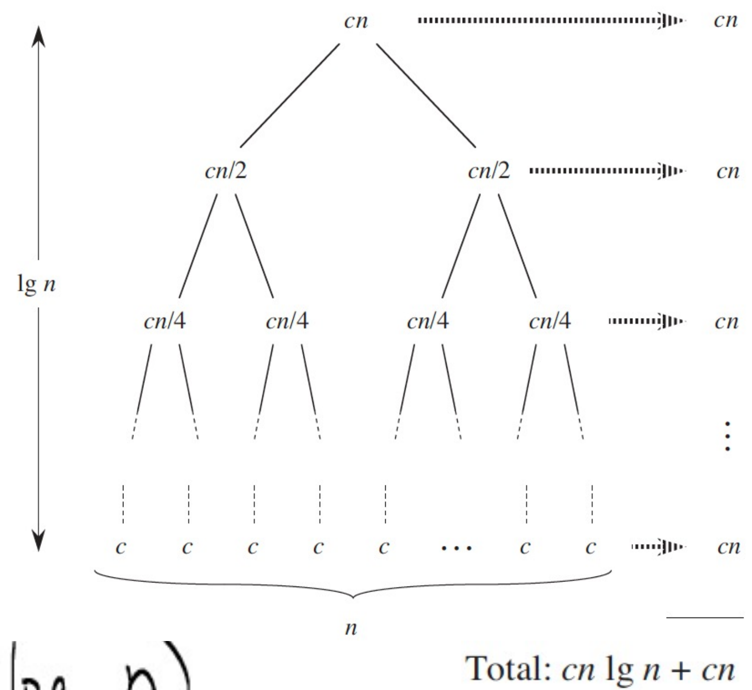
$$t(n) = n^d \underbrace{\sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i}_{\sum_{i=0}^{k} r^i}$$

Case 1 (r = 1) : a = b^d

e.g. mergesort

$$t(n) = a\, t\left(\frac{n}{b}\right) + cn^d$$

$a = 2$, $b = 2$, $d = 1$



$$t(n) = O(n^d \log_b n) = O(n \log_2 n)$$

The same amount of total work done at each level i, namely $O(n)$.

Case 2 ($r < 1$) : $a < b^d$ ☺

Here we have a decreasing amount of work at each level.

$$1 + r + r^2 + r^3 + \dots r^k$$

$$= \frac{1 - r^{k+1}}{1 - r}$$

$$< \frac{1}{1-r}, \quad \text{if } r < 1$$

$$= \text{constant (independent of } n)$$

$$\Rightarrow \quad t(n) = O(n^d)$$

$$t(n) = n^d \underbrace{\sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i}_{\sum_{i=0}^{k} r^i}$$

Case 3 ($r > 1$) : $\boxed{a > b^d}$

Here we have an increasing amount of work to do at each level.

The leaves dominate.

$$1 + r + r^2 + r^3 + \ldots r^k$$

$$= \frac{r^{k+1} - 1}{r - 1}$$

$< c r^k$, for some c which depends on r

$$t(n) = n^d \underbrace{\sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i}$$

$$\sum_{i=0}^{k} r^i$$

Case 3 (r > 1) : a > b^d

$$r^k = \left(\frac{a}{b^d}\right)^k$$

let $k = \log_b n$

$$= \left(\frac{a}{b^d}\right)^{\log_b n}$$

$$= a^{\log_b n} / (b^d)^{\log_b n}$$

$$= n^{\log_b a} / n^d$$

See Supplemental Slide

$$t(n) = n^d \underbrace{\sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i}_{\sum_{i=0}^{k} r^i}$$

Case 3 (r > 1) : a > b^d

$$t(n) = n^d \sum_{i=0}^{\log_b n} r^i$$

$$< n^d \, c \, r^{\log_b n}$$

$$= n^d \, c \left(\frac{a}{b^d}\right)^{\log_b n}$$

$$< n^d \cdot c \, \frac{n^{\log_b a}}{n^d}$$

$$= c \, n^{\log_b a}$$

$$t(n) = n^d \underbrace{\sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i}_{\sum_{i=0}^{k} r^i}$$

Master Method (summary)

$$t(n) = a\, t\left(\frac{n}{b}\right) + n^d, \quad t(1) = 1$$

same work at each level

$$t(n) \text{ is } \begin{cases} \boxed{O(n^d \log_b n)}, & \leftarrow a = b^d \\ \boxed{O(n^d)}, & \leftarrow a < b^d \\ \boxed{O(n^{\log_b a})}, & \leftarrow a > b^d \end{cases}$$

root dominates

leaves dominate

need to just identify your case

Master Method (summary)

$$t(n) = a\, t\left(\frac{n}{b}\right) + n^d, \quad t(1) = 1$$

same work at each level

$$t(n) \text{ is } \begin{cases} O(n^d \log_b n), & a = b^d \\ O(n^d), & a < b^d \\ O(n^{\log_b a}), & a > b^d \end{cases}$$

root dominates

leaves dominate

- Limitations:
  - Defined for worst case.
  - Sub-problems need to have the same size.
  - Applicable to recursions of divide-and-conquer solutions
  - Etc
  - Etc

Master theorem

- Goal: Recipe for solving common recurrences.

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

a = # subproblems

f(n) = work to divide/merge subproblems

b = factor by which the subproblem size decreases

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$. bad wins
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$. tie
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

good wins

Note that the three cases do not cover all the possibilities for $f(n)$.

Master theorem – Case 1

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers that satisfies the recurrence

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

where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Let $k = \log_b a$. Then,

Case 1. If $f(n) = O(n^{k-\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^k)$.

Ex. $T(n) = 3\,T(n/2) + n^{d}$.

- $a = 3,\ b = 2,\ f(n) = n,\ k = \log_2 3$.
- $T(n) = \Theta(n^{\lg 3})$.

case 1

The formula works with $\varepsilon = \log_2 3 - 1 > 0$

$$f(n) = n = O\left(n^{\log_2 3 - (\log_2 3 - 1)}\right)$$

Master theorem – Case 2

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers that satisfies the recurrence

$$T(n) = a\,T\left(\frac{n}{b}\right) \;+\; f(n)$$

where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Let $k = \log_b a$. Then,

Case 2. If $f(n) = \Theta(n^k \log^p n)$, then $T(n) = \Theta(n^k \log^{p+1} n)$.

Ex. $T(n) = 2\,T(n/2) + \Theta(n \log n)$.

- $a = 2,\ b = 2,\ f(n) = n \log n,\ k = \log_2 2 = 1,\ p = 1.$
- $T(n) = \Theta(n \log^2 n)$.

$$f(n) = \Theta(n \log n) = \Theta(n^{\log_2 2} \log n)$$

Master theorem – Case 3

Master theorem. Suppose that $T(n)$ is a function on the nonnegative integers that satisfies the recurrence

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

where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Let $k = \log_b a$. Then,

regularity condition holds if f(n) = $\Theta(n^{k+\varepsilon})$

Case 3. If $f(n) = \Omega(n^{k+\varepsilon})$ for some constant $\varepsilon > 0$ and if $a\,f(n/b) \le c\,f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Ex. $T(n) = 3\,T(n/4) + n^5$.

- $a = 3,\ b = 4,\ f(n) = n^5,\ k = \log_4 3$.
- $T(n) = \Theta(n^5)$.

1st property satisfied with $\varepsilon = 4 - \log_4 3$

$$f(n) = n^5 = \Omega(n^{\log_4 3 + (4 - \log_4 3)})$$

2nd property satisfied with $c = \frac{3}{4}$

$$3 \cdot \left(\frac{n}{4}\right)^5 \le c \cdot n^5$$

Master theorem – Applications

$k = \log_2 1 = 0; f(n) = 2^n$
$2^n = \Omega\left(n^{0+\log 2}\right)$
$1 \cdot 2^{\frac{n}{2}} \leq \frac{1}{2} \cdot 2^n$

$k = \log_2 3; f(n) = n^2$
$n^2 = \Omega\left(n^{\log_2 3 + (2 - \log_2 3)}\right)$
$3 \cdot \left(\frac{n}{2}\right)^2 \leq \frac{3}{4} \cdot n^2$

$T(n) = 3 * T(n/2) + n^2$
$\Rightarrow T(n) = \Theta(n^2)$ (case 3)

$T(n) = T(n/2) + 2^n$
$\Rightarrow T(n) = \Theta(2^n)$ (case 3)

$T(n) = 16 * T(n/4) + n$
$\Rightarrow T(n) = \Theta(n^2)$ (case 1)

$T(n) = 2 * T(n/2) + n \log n$
$\Rightarrow T(n) = n \log^2 n$ (case 2)

$T(n) = 2^n * T(n/2) + n^n$
$\Rightarrow$ Does not apply!!

$k = \log_4 16 = 2; f(n) = n$
$n = O(n^{2-1})$

$k = \log_2 2 = 1; f(n) = n \log n$
$n \log n = \Theta(n^1 log^1 n)$

Master theorem – Other variants – Akra-Bazzi

Desiderata. Generalizes master theorem to divide-and-conquer algorithms where subproblems have substantially different sizes.

Theorem. [Akra-Bazzi] Given constants $a_i > 0$ and $0 < b_i \leq 1$, functions $h_i(n) = O(n / \log^2 n)$ and $g(n) = O(n^c)$, if the function $T(n)$ satisfies the recurrence:

$$T(n) = \sum_{i=1}^{k} a_i T(b_i n + h_i(n)) + g(n)$$

a_i subproblems of size b_i n

small perturbation to handle floors and ceilings

Then $T(n) = \Theta\left(n^p\left(1+\int_1^n \frac{g(u)}{u^{p+1}}du\right)\right)$ where p satisfies $\sum_{i=1}^{k} a_i b_i^p = 1$.

Ex. $T(n) = 7/4\ T(\lfloor n/2 \rfloor) + T(\lceil 3/4\, n \rceil) + n^2$.

- $a_1 = 7/4,\ b_1 = 1/2,\ a_2 = 1,\ b_2 = 3/4 \Rightarrow p = 2$.
- $h_1(n) = \lfloor 1/2\, n \rfloor - 1/2\, n,\ h_2(n) = \lceil 3/4\, n \rceil - 3/4\, n$.
- $g(n) = n^2 \Rightarrow T(n) = \Theta(n^2 \log n)$.

Outline

- Complete Search
- Divide and Conquer.
  - Introduction.
  - Examples.
- Dynamic Programming.
- Greedy.


time for questions