Divide and Conquer

Arash Rafiey

27 October, 2016

- *Divide* the problem into a number of subproblems

- *Divide* the problem into a number of subproblems
- *Conquer* the subproblems by solving them recursively or if they are small, there must be an easy solution.

- *Divide* the problem into a number of subproblems
- *Conquer* the subproblems by solving them recursively or if they are small, there must be an easy solution.
- *Combine* the solutions to the subproblems to the solution of the problem

- *Divide* the problem into a number of subproblems
- *Conquer* the subproblems by solving them recursively or if they are small, there must be an easy solution.
- *Combine* the solutions to the subproblems to the solution of the problem

Example:

1. $\textsc{Merge-Sort}(A, p, r)$
2. **if** $p < r$
3. $\quad q := \lfloor (p + r)/2 \rfloor$
4. $\quad \textsc{Merge-Sort}(A, p, q)$
5. $\quad \textsc{Merge-Sort}(A, q + 1, r)$
6. $\quad \textsc{Merge}(A, p, q, r)$

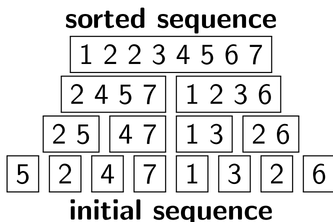
- *Divide* the problem into a number of subproblems
- *Conquer* the subproblems by solving them recursively or if they are small, there must be an easy solution.
- *Combine* the solutions to the subproblems to the solution of the problem

Example:

1. $\textsc{Merge-Sort}(A, p, r)$
2. **if** $p < r$
3. $\quad q := \lfloor (p + r)/2 \rfloor$
4. $\quad \textsc{Merge-Sort}(A, p, q)$
5. $\quad \textsc{Merge-Sort}(A, q + 1, r)$
6. $\quad \textsc{Merge}(A, p, q, r)$

Initial call for input array $A = A[1] \dots A[n]$ is $\textsc{Merge-Sort}(A, 1, n)$.

Example:



MERGE-SORT

- splits length-ℓ sequence into two length-$\ell/2$ sequences
- sorts them recursively
- merges the two sorted subsequences

Merge

$\textsc{Merge}(A, p, q, r)$

Take the smallest of the two front most elements of sequences $A[p..q]$ and $A[q+1..r]$ and put it into a temporary array.

Repeat this, until both sequences are empty. Copy the resulting sequence from temporary array into $A[p..r]$.

Merge

$\textsc{Merge}(A, p, q, r)$

Take the smallest of the two front most elements of sequences $A[p..q]$ and $A[q + 1..r]$ and put it into a temporary array.

Repeat this, until both sequences are empty. Copy the resulting sequence from temporary array into $A[p..r]$.

Write a pseudo code for the procedure $\textsc{Merge}(A, p, q, r)$ used in Merge-sort algorithm for merging two sorted arrays $A[p \dots q]$ and $A[q + 1 \dots r]$ into one.

MERGE(A, p, q, r)

1. $n_1 := q - p + 1$; $n_2 := r - q$;
2. **for** $i := 1$ **to** n_1
3. $\quad L[i] := A[p + i - 1]$
4. **for** $i := 1$ **to** n_2
5. $\quad R[i] := A[q + i]$
6. $i := 1$; $j := 1$; $k := p$
7. **while** $i \leq n_1$ and $j \leq n_2$
8. $\quad$ **if** $L[i] \leq R[j]$
9. $\quad\quad A[k] := L[i]$; $i := i + 1$;
10. $\quad$ **else**
11. $\quad\quad A[k] := R[j]$; $j := j + 1$;
12. $\quad k := k + 1$;
13. **if** $i > n_1$
14. $\quad$ **while** $j \leq n_2$
15. $\quad\quad A[k] := R[j]$; $j := j + 1$; $k := k + 1$;
16. **while** $i \leq n_1$
17. $\quad A[k] := L[i]$; $i := i + 1$; $k := k + 1$;

Analyzing an D&Q algorithm

Let $T(n)$ be running time on problem of size n

- If n is small enough (say, $n \leq c$ for constant c), then straightforward solution takes $\Theta(1)$

Analyzing an D&Q algorithm

Let $T(n)$ be running time on problem of size n

- If n is small enough (say, $n \leq c$ for constant c), then straightforward solution takes $\Theta(1)$
- If division of problem yields a subproblems, each of which $1/b$ of original (Merge-Sort: $a = b = 2$)

Analyzing an D&Q algorithm

Let $T(n)$ be running time on problem of size n

- If n is small enough (say, $n \leq c$ for constant c), then straightforward solution takes $\Theta(1)$
- If division of problem yields a subproblems, each of which $1/b$ of original (Merge-Sort: $a = b = 2$)
- Division into subproblems takes $D(n)$

Analyzing an D&Q algorithm

Let $T(n)$ be running time on problem of size n

- If n is small enough (say, $n \le c$ for constant c), then straightforward solution takes $\Theta(1)$
- If division of problem yields a subproblems, each of which $1/b$ of original (Merge-Sort: $a = b = 2$)
- Division into subproblems takes $D(n)$
- Combination of solutions to subproblems takes $C(n)$

Then,

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

For Merge-Sort:

- $a = b = 2$
- $D(n) = \Theta(1)$ (just compute "middle" of array)
- $C(n) = \Theta(n)$ (merging has running time linear in length of resulting sequence)

Thus

$$T_{\text{MS}}(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ T_{\text{MS}}(\lfloor n/2 \rfloor) + T_{\text{MS}}(\lceil n/2 \rceil) + \Theta(n) & \text{otherwise.} \end{cases}$$

That's what's called a **recurrence**

But: we want **closed form**, i.e., we want to *solve the recurrence*.

Solving recurrences

There are a few methods for solving recurrences, some easy and not powerful, some complicated and powerful.

Methods:

1. guess & verify (also called "*substitution method*")
2. *master* method
3. generating functions

and some others.

We're going to see 1. and 2.

Substitution method

Basic idea:

1. "*guess*" the form of the solution
2. Use mathematical induction to find constants and *show that solution works*.

Usually more difficult part is the part 1.
Back to example: we had

$$T_{\mathrm{MS}}(n) \leq 2T_{\mathrm{MS}}(\lceil n/2 \rceil) + \Theta(n)$$

for $n \geq 2$.
If you hadn't seen something like this before, **how would you guess?**
There is no general way to guess the correct solutions. It takes experience.

Heuristics that can help to find a good guess.

- One way would be to have a **look at first few terms.** Say if we had $T(n) = 2T(n/2) + 3n$, then

$$\begin{aligned} T(n) &= 2T(n/2) + 3n \\ &= 2(2T(n/4) + 3(n/2)) + 3n \\ &= 2(2(2T(n/8) + 3(n/4)) + 3(n/2)) + 3n \\ &= 2^3 T(n/2^3) + 2^2 3(n/2^2) + 2^1 3(n/2^1) + 2^0 3(n/2^0) \end{aligned}$$

  We can do this $\log n$ times

Heuristics that can help to find a good guess.

- One way would be to have a look at first few terms. Say if we had $T(n) = 2T(n/2) + 3n$, then

$$\begin{aligned} T(n) &= 2T(n/2) + 3n \\ &= 2(2T(n/4) + 3(n/2)) + 3n \\ &= 2(2(2T(n/8) + 3(n/4)) + 3(n/2)) + 3n \\ &= 2^3 T(n/2^3) + 2^2 3(n/2^2) + 2^1 3(n/2^1) + 2^0 3(n/2^0) \end{aligned}$$

We can do this $\log n$ times

$$\begin{aligned} & 2^{\log n} \cdot T(n/2^{\log n}) + \sum_{i=0}^{\log(n)-1} 2^i 3(n/2^i) \\ = \;& n \cdot T(1) + 3n \cdot \sum_{i=0}^{\log(n)-1} 1 \\ = \;& n \cdot T(1) + 3n \log n = \Theta(n \log n) \end{aligned}$$

After guessing a solution you'll have to **prove the correctness**.

- **similar recurrences** might have similar solutions
  Consider
  $$T(n) = 2T(\lfloor n/2 \rfloor + 25) + n$$
  Looks similar to last example, but is the additional 25 in the **argument** going to change the solution?
  Not really, because for large n, difference between
  $$T(\lfloor n/2 \rfloor) \quad \text{and} \quad T(\lfloor n/2 \rfloor + 25)$$
  is not large: both cut n nearly in half: for $n = 2,000$ we have
  $$T(1,000) \quad \text{and} \quad T(1,025),$$
  for $n = 1,000,000$ we have
  $$T(500,000) \quad \text{and} \quad T(500,025).$$
  Thus, reasonable assumption is that now
  $T(n) = O(n \log n)$ as well.

- **stepwise refinement** – guessing loose lower and upper bounds, and gradually taking them closer to each other
  For $T(n) = 2T(\lfloor n/2 \rfloor) + n$ we see

- **stepwise refinement** – guessing loose lower and upper bounds, and gradually taking them closer to each other
  For $T(n) = 2T(\lfloor n/2 \rfloor) + n$ we see
  - $T(n) = \Omega(n)$ (because of the n term)
  - $T(n) = O(n^2)$ (easily proven)

- **stepwise refinement** – guessing loose lower and upper bounds, and gradually taking them closer to each other
  For $T(n) = 2T(\lfloor n/2 \rfloor) + n$ we see
  - $T(n) = \Omega(n)$ (because of the n term)
  - $T(n) = O(n^2)$ (easily proven)

  From there, we can perhaps "converge" on asymptotically tight bound $\Theta(n \log n)$.

Proving correctness of a guess

For Merge-Sort we have $T(n) \leq 2T(\lceil n/2 \rceil) + \Theta(n)$, which means, there is a constant $d > 0$ such that

$$T(n) \leq 2T(\lceil n/2 \rceil) + dn \qquad \text{for } n \geq 2.$$

We guessed that $T(n) = O(n \log n)$.
We prove $T(n) \leq cn \log n$ for appropriate choice of constant $c > 0$.

Induction hypothesis: assume that bound holds for any $n < m$, hence also for $n = \lceil m/2 \rceil$, i.e.,

$$T(\lceil m/2 \rceil) \leq c \lceil m/2 \rceil \log(\lceil m/2 \rceil)$$

and prove it for $m \geq 4$ (**induction step**):

$$T(m) \leq 2T(\lceil m/2 \rceil) + dm$$

and prove it for $m \geq 4$ (**induction step**):

$$
\begin{aligned}
T(m) &\leq 2T(\lceil m/2 \rceil) + dm \\
&\leq 2(c\lceil m/2 \rceil \log(\lceil m/2 \rceil)) + dm
\end{aligned}
$$

and prove it for $m \geq 4$ (**induction step**):

$$
\begin{aligned}
T(m) &\leq 2T(\lceil m/2 \rceil) + dm \\
&\leq 2(c\lceil m/2 \rceil \log(\lceil m/2 \rceil)) + dm \\
&\leq 2(c(m+1)/2 \cdot \log((m+1)/2)) + dm
\end{aligned}
$$

and prove it for $m \geq 4$ (**induction step**):

$$\begin{aligned} T(m) &\leq 2T(\lceil m/2 \rceil) + dm \\ &\leq 2(c\lceil m/2 \rceil \log(\lceil m/2 \rceil)) + dm \\ &\leq 2(c(m+1)/2 \cdot \log((m+1)/2)) + dm \\ &\leq c(m+1)\log((m+1)/2) + dm \end{aligned}$$

and prove it for $m \geq 4$ (**induction step**):

$$\begin{aligned}
T(m) &\leq 2T(\lceil m/2 \rceil) + dm \\
&\leq 2(c\lceil m/2 \rceil \log(\lceil m/2 \rceil)) + dm \\
&\leq 2(c(m+1)/2 \cdot \log((m+1)/2)) + dm \\
&\leq c(m+1)\log((m+1)/2) + dm \\
&= c(m+1)\log(m+1) - c(m+1)\log 2 + dm
\end{aligned}$$

and prove it for $m \geq 4$ (**induction step**):

$$\begin{aligned}
T(m) &\leq 2T(\lceil m/2 \rceil) + dm \\
&\leq 2(c\lceil m/2 \rceil \log(\lceil m/2 \rceil)) + dm \\
&\leq 2(c(m+1)/2 \cdot \log((m+1)/2)) + dm \\
&\leq c(m+1)\log((m+1)/2) + dm \\
&= c(m+1)\log(m+1) - c(m+1)\log 2 + dm \\
&\leq c(m+1)(\log m + 1/3) - c(m+1) + dm
\end{aligned}$$

and prove it for $m \geq 4$ (**induction step**):

$$
\begin{aligned}
T(m) &\leq 2T(\lceil m/2 \rceil) + dm \\
&\leq 2(c\lceil m/2 \rceil \log(\lceil m/2 \rceil)) + dm \\
&\leq 2(c(m+1)/2 \cdot \log((m+1)/2)) + dm \\
&\leq c(m+1)\log((m+1)/2) + dm \\
&= c(m+1)\log(m+1) - c(m+1)\log 2 + dm \\
&\leq c(m+1)(\log m + 1/3) - c(m+1) + dm \\
&= cm\log m + c\log m + c/3(m+1) - c(m+1) + dm
\end{aligned}
$$

and prove it for $m \geq 4$ (**induction step**):

$$\begin{aligned}
T(m) &\leq 2T(\lceil m/2 \rceil) + dm \\
&\leq 2(c\lceil m/2 \rceil \log(\lceil m/2 \rceil)) + dm \\
&\leq 2(c(m+1)/2 \cdot \log((m+1)/2)) + dm \\
&\leq c(m+1)\log((m+1)/2) + dm \\
&= c(m+1)\log(m+1) - c(m+1)\log 2 + dm \\
&\leq c(m+1)(\log m + 1/3) - c(m+1) + dm \\
&= cm\log m + c\log m + c/3(m+1) - c(m+1) + dm \\
&\leq cm\log m + cm/2 - 2/3c(m+1) + dm
\end{aligned}$$

and prove it for $m \geq 4$ (**induction step**):

$$
\begin{aligned}
T(m) &\leq 2T(\lceil m/2 \rceil) + dm \\
&\leq 2(c\lceil m/2 \rceil \log(\lceil m/2 \rceil)) + dm \\
&\leq 2(c(m+1)/2 \cdot \log((m+1)/2)) + dm \\
&\leq c(m+1)\log((m+1)/2) + dm \\
&= c(m+1)\log(m+1) - c(m+1)\log 2 + dm \\
&\leq c(m+1)(\log m + 1/3) - c(m+1) + dm \\
&= cm\log m + c\log m + c/3(m+1) - c(m+1) + dm \\
&\leq cm\log m + cm/2 - 2/3c(m+1) + dm \\
&\leq cm\log m + (c/2 - 2/3c + d)m = cm\log m + (d - c/6)m
\end{aligned}
$$

and prove it for $m \geq 4$ (**induction step**):

$$
\begin{aligned}
T(m) &\leq 2T(\lceil m/2 \rceil) + dm \\
&\leq 2(c\lceil m/2 \rceil \log(\lceil m/2 \rceil)) + dm \\
&\leq 2(c(m+1)/2 \cdot \log((m+1)/2)) + dm \\
&\leq c(m+1)\log((m+1)/2) + dm \\
&= c(m+1)\log(m+1) - c(m+1)\log 2 + dm \\
&\leq c(m+1)(\log m + 1/3) - c(m+1) + dm \\
&= cm\log m + c\log m + c/3(m+1) - c(m+1) + dm \\
&\leq cm\log m + cm/2 - 2/3c(m+1) + dm \\
&\leq cm\log m + (c/2 - 2/3c + d)m = cm\log m + (d - c/6)m \\
&\leq cm\log m \quad \text{for } c \geq 6d.
\end{aligned}
$$

Base step

It remains to show that boundary conditions of recurrence ($m < 4$) are **suitable as base cases for the inductive proof**.
We have got to show that we can choose c large enough s.t.bound

$$T(m) \leq cm \log m$$

works for boundary conditions as well (when $m < 4$).
Assume that $T(1) = b > 0$.
For $m = 1$,

$$T(m) \leq cm \log m = c \cdot 1 \cdot 0 = 0$$

is a bit of a problem, because $T(1)$ is a constant greater than 0.

How to resolve this problem?
Recall that we wanted to prove that $T(n) = O(n \log n)$.
Also, recall that by def of $O()$, we are free to disregard a constant number of small values of n: $f(n) = O(g(n)) \Leftrightarrow$
there exist constants $c, n_0 :\ f(n) \leq c \cdot g(n)$ for $n \geq n_0$
A way out of our problem is to **remove** difficult boundary condition $T(1) = b > 0$ from consideration in inductive proof.

Note: for $m \geq 4$, $T(m)$ does **not** depend directly on $T(1)$
$(T(2) \leq 2T(1) + 2d = 2b + 2d,$
$T(3) \leq 2T(2) + 3d = 2(2b + 2d) + 3d = 4b + 7d,$
$T(4) \leq 2T(2) + 4d)$

This means:
We replace $T(1)$ by $T(2)$ and $T(3)$ as the base case in the inductive proof, letting $n_0 = 2$.

In other words, we are showing that the bound

$$T(n) \leq cn \log n$$

holds for any $n \geq 2$.

- For $m = 2$: $T(2) = 2b + 2d? \leq ?c.2 \log 2 = 2c$
- For $m = 3$: $T(3) = 4b + 7d? \leq ?c.3 \log 3$

Hence, set c to $\max(6d, b + d, \frac{4b+7d}{3 \log 3})$ and the above boundary conditions as well as the induction step will work.

Important:
General technique! Can be used very often!

A neat trick called “changing variables”

Suppose we have

$$T(n) = 2T(\sqrt{n}) + \log n$$

A neat trick called “changing variables”

Suppose we have

$$T(n) = 2T(\sqrt{n}) + \log n$$

Now rename $m = \log n \Leftrightarrow 2^m = n$. We know $\sqrt{n} = n^{1/2} = (2^m)^{1/2} = 2^{m/2}$ and thus obtain

$$T(2^m) = 2T(2^{m/2}) + m$$

A neat trick called "changing variables"

Suppose we have

$$T(n) = 2T(\sqrt{n}) + \log n$$

Now rename $m = \log n \Leftrightarrow 2^m = n$. We know $\sqrt{n} = n^{1/2} = (2^m)^{1/2} = 2^{m/2}$ and thus obtain

$$T(2^m) = 2T(2^{m/2}) + m$$

Now rename $S(m) = T(2^m)$ and get

$$S(m) = 2S(m/2) + m$$

Looks familiar. We know the solution $S(m) = \Theta(m \log m)$.

A neat trick called "changing variables"

Suppose we have

$$T(n) = 2T(\sqrt{n}) + \log n$$

Now rename $m = \log n \Leftrightarrow 2^m = n$. We know $\sqrt{n} = n^{1/2} = (2^m)^{1/2} = 2^{m/2}$ and thus obtain

$$T(2^m) = 2T(2^{m/2}) + m$$

Now rename $S(m) = T(2^m)$ and get

$$S(m) = 2S(m/2) + m$$

Looks familiar. We know the solution $S(m) = \Theta(m \log m)$. Going back from $S(m)$ to $T(n)$ we obtain

$$T(n) = T(2^m) = S(m) = \Theta(m \log m) = \Theta(\log n \log \log n)$$

Some basic algebra

Sums

$a + (a+1) + (a+2) + \cdots + (b-1) + b = \sum_{i=a}^{b} i = \frac{(a+b)(b-a+1)}{2}$

Some basic algebra
Sums

$$a + (a+1) + (a+2) + \cdots + (b-1) + b = \sum_{i=a}^{b} i = \frac{(a+b)(b-a+1)}{2}$$

$$a + a \cdot c + a \cdot c^2 + \cdots + a \cdot c^{n-1} + a \cdot c^n = \sum_{i=0}^{n} a \cdot c^i = a \cdot \frac{1-c^{n+1}}{1-c}$$

if $0 < c < 1$ then we can estimate the sum as follows

Some basic algebra
Sums

$a + (a+1) + (a+2) + \cdots + (b-1) + b = \sum_{i=a}^{b} i = \frac{(a+b)(b-a+1)}{2}$

$a + a \cdot c + a \cdot c^2 + \cdots + a \cdot c^{n-1} + a \cdot c^n = \sum_{i=0}^{n} a \cdot c^i = a \cdot \frac{1-c^{n+1}}{1-c}$

if $0 < c < 1$ then we can estimate the sum as follows

$\sum_{i=0}^{n} a \cdot c^i = a \cdot \frac{1-c^{n+1}}{1-c} < a \cdot \frac{1}{1-c}$

logarithms and powers

$\log_a n$ and a^n are inverse functions to each other:

logarithms and powers

$\log_a n$ and a^n are inverse functions to each other:

$\boxed{\log_a(a^n) = n}$ for all n

logarithms and powers

$\log_a n$ and a^n are inverse functions to each other:

$\boxed{\log_a(a^n) = n}$ for all n

$\boxed{a^{\log_a n} = n}$ for all $n > 0$

logarithms and powers

$\log_a n$ and a^n are inverse functions to each other:

$\boxed{\log_a(a^n) = n}$ for all n

$\boxed{a^{\log_a n} = n}$ for all $n > 0$

properties:

$\log_a(b \cdot c) = \log_a b + \log_a c$

logarithms and powers

$\log_a n$ and a^n are inverse functions to each other:

$\boxed{\log_a(a^n) = n}$ for all n

$\boxed{a^{\log_a n} = n}$ for all $n > 0$

properties:

$\log_a(b \cdot c) = \log_a b + \log_a c$

$\log_a b = \frac{\log_c b}{\log_c a}$

logarithms and powers

$\log_a n$ and a^n are inverse functions to each other:

$\boxed{\log_a(a^n) = n}$ for all n

$\boxed{a^{\log_a n} = n}$ for all $n > 0$

properties:

$\log_a(b \cdot c) = \log_a b + \log_a c$

$\log_a b = \frac{\log_c b}{\log_c a}$

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

logarithms and powers

$\log_a n$ and a^n are inverse functions to each other:

$\boxed{\log_a(a^n) = n}$ for all n

$\boxed{a^{\log_a n} = n}$ for all $n > 0$

properties:

$\log_a(b \cdot c) = \log_a b + \log_a c$

$\log_a b = \frac{\log_c b}{\log_c a}$

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

$a^{\log_c b} = b^{\log_c a}$

The "Master Method"

Recipe for recurrences of the form

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

with $a \geq 1$ and $b > 1$ constant, and $f(n)$ an asymptotically positive function ($f(n) = 5$, $f(n) = c \log n$, $f(n) = n$, $f(n) = n^{12}$ are just fine).

Split problem into a subproblems each of size n/b.

Subproblems are solved recursively, each in time $T(n/b)$.

Dividing problem and combining solutions of subproblems is captured by $f(n)$.

Deals with many frequently seen recurrences (in particular, our Merge-Sort example with $a = b = 2$ and $f(n) = \Theta(n)$).

Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. *If $f(n) = \mathcal{O}(n^{(\log_b a)-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.*
2. *If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.*
3. *If $f(n) = \Omega(n^{(\log_b a)+\epsilon})$ for some constant $\epsilon > 0$, and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

Notes on Master's Theorem

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.

Note 1: Although it's looking rather scary, it really isn't. For instance, with Merge-Sort's recurrence $T(n) = 2T(n/2) + \Theta(n)$ we have $n^{\log_b a} = n^{\log_2 2} = n^1 = n$, and we can apply case 2. The result is therefore $\Theta(n^{\log_b a} \cdot \log n) = \Theta(n \log n)$.

Notes on Master's Theorem

2. If $f(n) = \Theta(n^{\log_b a})$, then
   $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.

Note 1: Although it's looking rather scary, it really isn't. For instance, with Merge-Sort's recurrence $T(n) = 2T(n/2) + \Theta(n)$ we have $n^{\log_b a} = n^{\log_2 2} = n^1 = n$, and we can apply case 2. The result is therefore $\Theta(n^{\log_b a} \cdot \log n) = \Theta(n \log n)$.

1. If $f(n) = \mathcal{O}(n^{(\log_b a)-\epsilon})$ for some constant $\epsilon > 0$, then
   $T(n) = \Theta(n^{\log_b a})$.

Note 2: In case 1,

$$f(n) = n^{(\log_b a)-\epsilon} = n^{\log_b a}/n^{\epsilon} = o(n^{\log_b a}),$$

so the ϵ **does** matter. This case is basically about "small" functions f. But it's not enough if $f(n)$ is just asymptotically smaller than $n^{\log_b a}$ (that is $f(n) \in o(n^{\log_b a})$), it must be *polynomially smaller*!

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

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

Note 3: Similarly, in case 3,

$$f(n) = n^{(\log_b a)+\epsilon} = n^{\log_b a} \cdot n^{\epsilon} = \omega(n^{\log_b a}),$$

so the ϵ **does** matter again. This case is basically about "large" functions n. But again, $f(n) \in \omega(n^{\log_b a})$ is not enough, it must be *polynomially larger*. And in addition $f(n)$ has to satisfy the "**regularity condition**":

$$af(n/b) \leq cf(n)$$

for some constant $c < 1$ and $n \geq n_0$ for some n_0.

The idea is that we *compare* $n^{\log_b a}$ to $f(n)$.

Result is (intuitively) determined by larger of two.

The idea is that we *compare* $n^{\log_b a}$ to $f(n)$.

Result is (intuitively) determined by larger of two.

In case 1, $n^{\log_b a}$ is larger, so result is $\Theta(n^{\log_b a})$.

The idea is that we *compare* $n^{\log_b a}$ to $f(n)$.

Result is (intuitively) determined by larger of two.

In case 1, $n^{\log_b a}$ is larger, so result is $\Theta(n^{\log_b a})$.

In case 3, $f(n)$ is larger, so result is $\Theta(f(n))$.

The idea is that we *compare* $n^{\log_b a}$ to $f(n)$.

Result is (intuitively) determined by larger of two.

In case 1, $n^{\log_b a}$ is larger, so result is $\Theta(n^{\log_b a})$.

In case 3, $f(n)$ is larger, so result is $\Theta(f(n))$.

In case 2, both have same order, we multiply it by a logarithmic factor, and result is $\Theta(n^{\log_b a} \cdot \log n) = \Theta(f(n) \cdot \log n)$.

The idea is that we *compare* $n^{\log_b a}$ to $f(n)$.

Result is (intuitively) determined by larger of two.

In case 1, $n^{\log_b a}$ is larger, so result is $\Theta(n^{\log_b a})$.

In case 3, $f(n)$ is larger, so result is $\Theta(f(n))$.

In case 2, both have same order, we multiply it by a logarithmic factor, and result is $\Theta(n^{\log_b a} \cdot \log n) = \Theta(f(n) \cdot \log n)$.

Important: Does not cover **all** possible cases.

For instance, there is a gap between cases 1 and 2 whenever $f(n)$ is smaller than $n^{\log_b a}$ but not **polynomially** smaller.

Using the master theorem

Simple enough. Some examples:

$$T(n) = 9T(n/3) + n$$

We have $a = 9$, $b = 3$, $f(n) = n$. Thus, $n^{\log_b a} = n^{\log_3 9} = n^2$. Clearly, $f(n) = \mathcal{O}(n^{\log_3(9)-\epsilon})$ for $\epsilon = 1$, so case 1 gives $T(n) = \Theta(n^2)$.

Using the master theorem

Simple enough. Some examples:

$T(n) = 9T(n/3) + n$

We have $a = 9$, $b = 3$, $f(n) = n$. Thus, $n^{\log_b a} = n^{\log_3 9} = n^2$. Clearly, $f(n) = \mathcal{O}(n^{\log_3(9)-\epsilon})$ for $\epsilon = 1$, so case 1 gives $T(n) = \Theta(n^2)$.

$T(n) = T(2n/3) + 1$

We have $a = 1$, $b = 3/2$, and $f(n) = 1$, so $n^{\log_b a} = n^{\log_{2/3} 1} = n^0 = 1$.

Apply case 2 ($f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, result is $T(n) = \Theta(\log n)$.

$T(n) = 3T(n/4) + n \log n$

We have $a = 3$, $b = 4$, and $f(n) = n \log n$, so $n^{\log_b a} = n^{\log_4 3} = \mathcal{O}(n^{0.793})$.

Clearly, $f(n) = n \log n = \Omega(n)$ and

thus also $f(n) = \Omega(n^{\log_b(a)+\epsilon})$

for $\epsilon \approx 0.2$. Also, $a \cdot f(n/b) = 3(n/4)\log(n/4) \leq (3/4)n \log n = c \cdot f(n)$ for any $c = 3/4 < 1$.

Thus we can apply case 3 with result $T(n) = \Theta(n \log n)$.

Exercises

Problem

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$.

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

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

(c) $T(n) = T(\sqrt{n}) + \log\log n$

(d) $T(n) = 16T(n/4) + n^2$

(e) $T(n) = 2T(n-1) + \log n$

Integer Multiplication

We are given two n-bits numbers x, y and we want to compute $x \times y$.

Integer Multiplication

We are given two n-bits numbers x, y and we want to compute $x \times y$.

The usual multiplication takes $\mathcal{O}(n^2)$ times.

Integer Multiplication

We are given two n-bits numbers x, y and we want to compute $x \times y$.

The usual multiplication takes $\mathcal{O}(n^2)$ times.

Can we do better than $\mathcal{O}(n^2)$?

Integer Multiplication

We are given two n-bits numbers x, y and we want to compute $x \times y$.

The usual multiplication takes $\mathcal{O}(n^2)$ times.

Can we do better than $\mathcal{O}(n^2)$?

We can write $x = x_1 2^{\frac{n}{2}} + x_0$ and $y = y_1 2^{\frac{n}{2}} + y_0$.

x_0, x_1, y_0, y_1 are $\frac{n}{2}$-bits numbers.

For simplicity we write xy instead of $x \times y$.

$$xy = (x_1 2^{\frac{n}{2}} + x_0)(y_1 2^{\frac{n}{2}} + y_0) = x_1 y_1 2^n + (x_1 y_0 + x_0 y_1) 2^{\frac{n}{2}} + x_0 y_0.$$

For simplicity we write xy instead of $x \times y$.

$$xy = (x_1 2^{\frac{n}{2}} + x_0)(y_1 2^{\frac{n}{2}} + y_0) = x_1 y_1 2^n + (x_1 y_0 + x_0 y_1) 2^{\frac{n}{2}} + x_0 y_0.$$

Now if we compute each of the $x_1 y_1, x_0 y_0, x_1 y_0, x_0 y_1$ then we can compute xy.

For simplicity we write xy instead of $x \times y$.

$$xy = (x_1 2^{\frac{n}{2}} + x_0)(y_1 2^{\frac{n}{2}} + y_0) = x_1 y_1 2^n + (x_1 y_0 + x_0 y_1) 2^{\frac{n}{2}} + x_0 y_0.$$

Now if we compute each of the $x_1 y_1, x_0 y_0, x_1 y_0, x_0 y_1$ then we can compute xy.

$T(n) \leq 4T(n/2) + cn.$

$O(n)$ is the time takes to add two n-bits numbers.

For simplicity we write xy instead of $x \times y$.

$xy = (x_1 2^{\frac{n}{2}} + x_0)(y_1 2^{\frac{n}{2}} + y_0) = x_1 y_1 2^n + (x_1 y_0 + x_0 y_1) 2^{\frac{n}{2}} + x_0 y_0$.

Now if we compute each of the $x_1 y_1, x_0 y_0, x_1 y_0, x_0 y_1$ then we can compute xy.

$T(n) \leq 4T(n/2) + cn$.

$O(n)$ is the time takes to add two n-bits numbers.

Using the Master Method for $a = 4, b = 2$ and $f(n) = cn$ we obtain :

$T(n) \leq \mathcal{O}(n^{\log_2 4}) = \mathcal{O}(n^2)$.

Not good!!

If we compute $p = (x_1 + x_0)(y_1 + y_0) = x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0$ then instead of $x_1 y_0 + x_0 y_1$ we can write $p - x_1 y_1 - x_0 y_0$.

If we compute $p = (x_1 + x_0)(y_1 + y_0) = x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0$ then instead of $x_1 y_0 + x_0 y_1$ we can write $p - x_1 y_1 - x_0 y_0$.

p is the multiplication of two $\frac{n}{2}$-bits numbers.

So we need to compute the multiplication of three pairs of $\frac{n}{2}$-bits numbers.

Recursive-Multiply(x,y)

1. $x := x_1 2^{\frac{n}{2}} + x_0$ and $y := y_1 2^{\frac{n}{2}} + y_0$.
2. Compute $x_1 + x_0$ and $y_1 + y_0$
3. $p :=$ Recursive-Multiply $(x_0 + x_1, y_0 + y_1)$
4. $x_1 y_1 :=$ Recursive-Multiply (x_1, y_1)
5. $x_0 y_0 :=$ Recursive-Multiply (x_0, y_0)
6. Return $x_1 y_1 2^n + (p - x_1 y_1 - x_0 y_0) 2^{\frac{n}{2}} + x_0 y_0$

Recursive-Multiply(x,y)

1. $x := x_1 2^{\frac{n}{2}} + x_0$ and $y := y_1 2^{\frac{n}{2}} + y_0$.
2. Compute $x_1 + x_0$ and $y_1 + y_0$
3. $p :=$ Recursive-Multiply $(x_0 + x_1, y_0 + y_1)$
4. $x_1 y_1 :=$ Recursive-Multiply (x_1, y_1)
5. $x_0 y_0 :=$ Recursive-Multiply (x_0, y_0)
6. Return $x_1 y_1 2^n + (p - x_1 y_1 - x_0 y_0) 2^{\frac{n}{2}} + x_0 y_0$

$T(n) \leq 3T(n/2) + cn.$
$T(n) \leq \mathcal{O}(n^{\log_2 3}) = \mathcal{O}(n^{1.59}).$

Smallest distance between pairs of points

We are give a set $\{p_1, p_2, \ldots, p_n\}$ of n points in plane (2D plane). Find a pair of points with the smallest distance.

Smallest distance between pairs of points

We are give a set $\{p_1, p_2, \dots, p_n\}$ of n points in plane (2D plane). Find a pair of points with the smallest distance.

It is clear that we can solve the problem in $\mathcal{O}(n^2)$.

Can we do better than $\mathcal{O}(n^2)$?

Smallest distance between pairs of points

We are give a set $\{p_1, p_2, \ldots, p_n\}$ of n points in plane (2D plane). Find a pair of points with the smallest distance.

It is clear that we can solve the problem in $\mathcal{O}(n^2)$.

Can we do better than $\mathcal{O}(n^2)$?

Sort the points by x-coordinate and produce list P_x ($\mathcal{O}(n \log n)$).

Smallest distance between pairs of points

We are give a set $\{p_1, p_2, \ldots, p_n\}$ of n points in plane (2D plane). Find a pair of points with the smallest distance.

It is clear that we can solve the problem in $\mathcal{O}(n^2)$.

Can we do better than $\mathcal{O}(n^2)$?

Sort the points by x-coordinate and produce list P_x ($\mathcal{O}(n \log n)$).

Sort the points by y-coordinate and produce list P_y ($\mathcal{O}(n \log n)$).

Smallest distance between pairs of points

We are give a set $\{p_1, p_2, \ldots, p_n\}$ of n points in plane (2D plane). Find a pair of points with the smallest distance.

It is clear that we can solve the problem in $\mathcal{O}(n^2)$.

Can we do better than $\mathcal{O}(n^2)$?

Sort the points by x-coordinate and produce list P_x ($\mathcal{O}(n \log n)$).

Sort the points by y-coordinate and produce list P_y ($\mathcal{O}(n \log n)$).

Set Q be the set of points in the first $\frac{n}{2}$ positions in P_x and let R be the rest of the points in P_x.

Construct the lists Q_x and Q_y and R_x and R_y.

Smallest distance between pairs of points

We are give a set $\{p_1, p_2, \ldots, p_n\}$ of n points in plane (2D plane). Find a pair of points with the smallest distance.

It is clear that we can solve the problem in $\mathcal{O}(n^2)$.

Can we do better than $\mathcal{O}(n^2)$?

Sort the points by x-coordinate and produce list P_x ($\mathcal{O}(n \log n)$).

Sort the points by y-coordinate and produce list P_y ($\mathcal{O}(n \log n)$).

Set Q be the set of points in the first $\frac{n}{2}$ positions in P_x and let R be the rest of the points in P_x.

Construct the lists Q_x and Q_y and R_x and R_y.

1. Recursively compute closest pair of points for Q, say q_0, q_1 and

Smallest distance between pairs of points

We are give a set $\{p_1, p_2, \ldots, p_n\}$ of n points in plane (2D plane). Find a pair of points with the smallest distance.

It is clear that we can solve the problem in $\mathcal{O}(n^2)$.

Can we do better than $\mathcal{O}(n^2)$?

Sort the points by x-coordinate and produce list P_x ($\mathcal{O}(n \log n)$).

Sort the points by y-coordinate and produce list P_y ($\mathcal{O}(n \log n)$).

Set Q be the set of points in the first $\frac{n}{2}$ positions in P_x and let R be the rest of the points in P_x.

Construct the lists Q_x and Q_y and R_x and R_y.

1. Recursively compute closest pair of points for Q, say q_0, q_1 and
2. Recursively compute closest pair of points for R, say r_0, r_1.

Let δ be the smallest of $d(q_0, q_1), d(r_0, r_1)$ ($d(x, y)$ denote the distance between x, y).

Let x^* denote the coordinates of the rightmost point in Q and let L be the line $x^* = x$.
L separates Q from R.

Let δ be the smallest of $d(q_0, q_1), d(r_0, r_1)$ ($d(x, y)$ denote the distance between x, y).

Let x^* denote the coordinates of the rightmost point in Q and let L be the line $x^* = x$.
L separates Q from R.

If there exist $q \in Q$ and $r \in R$ with $d(q, r) < \delta$ then each of q, r lies in distance at most δ of L.

Let δ be the smallest of $d(q_0, q_1), d(r_0, r_1)$ ($d(x, y)$ denote the distance between x, y).

Let x^* denote the coordinates of the rightmost point in Q and let L be the line $x^* = x$.
L separates Q from R.

If there exist $q \in Q$ and $r \in R$ with $d(q, r) < \delta$ then each of q, r lies in distance at most δ of L.

So we can narrow down our search...
Let S be the set of points in P in distance δ from L.

Let δ be the smallest of $d(q_0, q_1), d(r_0, r_1)$ ($d(x, y)$ denote the distance between x, y).

Let x^* denote the coordinates of the rightmost point in Q and let L be the line $x^* = x$.
L separates Q from R.

If there exist $q \in Q$ and $r \in R$ with $d(q, r) < \delta$ then each of q, r lies in distance at most δ of L.

So we can narrow down our search...
Let S be the set of points in P in distance δ from L.

Let S_y denote the list of points in S sorted by increasing y-coordinate. (by going through list P_y, we can compute S_y in $\mathcal{O}(n)$.)

There exist $q \in Q$ and $r \in R$ for which $d(q, r) < \delta$ if and only if there exist $s, s' \in S$ for which $d(s, s') < \delta$.

There exist $q \in Q$ and $r \in R$ for which $d(q, r) < \delta$ if and only if there exist $s, s' \in S$ for which $d(s, s') < \delta$.

Now partition S into grids of size $\delta/2$ by $\delta/2$ in such a way that L is one of the lines of the grid.

We note that in each of the grid cells there is at most one point. Otherwise the distance between two points in that grid cell is at most $\delta\sqrt{2}/2 < \delta$. A contradiction to $\delta = \min\{d(q_0, q_1), d(r_0, r_1)\}$.

If $s, s' \in S$ have the property that $d(s, s') < \delta$ then s, s' are within 15 positions of each other in the sorted list S_y.

ClosestPair(P)

1. Construct P_x and P_y ($\mathcal{O}(n \log n)$ time)
2. $(p_0^*, p_1^*) =$ Closest-Pair(P_x, P_y)

Closest-Pair(P_x, P_y)

1. If $|P| \leq 3$ compute the closest pairs by trying all and return...
2. Construct Q_x, Q_y and R_x, R_y ($\mathcal{O}(n)$ time)
3. $(q_0^*, q_1^*) = Closest - Pair(Q_x, Q_y)$
4. $(r_0^*, r_1^*) = Closest - Pair(R_x, R_y)$
5. $\delta := \min(d(q_0^*, q_1^*), d(r_0^*, r_1^*))$.

6.$x^* =$ maximum x-coordinate of a point in set Q

7. $L = \{(x, y) : x = x^*\}$
8. $S =$ points in P within distance δ of L
9. Construct S_y

10. **for** each point $s \in S_y$, compute the distance from s to each of next 15 points in S_y.

11. Let s, s' be pair achieving minimum distance ($\mathcal{O}(n)$ time)

12. **if** $d(s, s') < \delta$ return (s, s')

13. **if** $d(q_0^*, q_1^*) < d(r_0^*, r_1^*))$ return (q_0^*, q_1^*)

14. **else** return (r_0^*, r_1^*)

Matrix Multiplication

Given : We are given two $n \times n$ matrixes A, B of integers.

Goal : We want to compute $A \times B$.

- Direct approach: has $\mathcal{O}(n^3)$ time complexity.
- Divide and Conquer:

$$C_{n\times n} = A_{n\times n} \times B_{n\times n}$$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$n/2$

$n/2$

- $C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$
- $C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$
- $C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$
- $C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$

$$C_{n\times n} = A_{n\times n} \times B_{n\times n}$$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

(each block is $n/2 \times n/2$)

- $C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$
- $C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$
- $C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$
- $C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$

In this case we have $T(n) = 8T(n/2) + \mathcal{O}(n^2) = \Theta(n^3)$
How we can reduce the time complexity?

Strassen's Matrix Multiplication

In order to improve the algorithm we have to reduce the number of multiplication by introducing the new variables as follows:

- $P = (A_{11} + A_{22}) \times (B_{11} + B_{22})$
- $Q = (A_{21} + A_{22}) \times B_{11}$
- $R = A_{11} \times (B_{12} - B_{22})$
- $S = A_{22} \times (B_{21} - B_{11})$
- $T = (A_{11} + A_{12}) \times B_{22}$
- $U = (A_{21} - A_{11}) \times (B_{11} + B_{12})$
- $V = (A_{12} - A_{22}) \times (B_{21} + B_{22})$

Now, we have:

$C_{11} = P + S - T + V$
$C_{12} = R + T$
$C_{21} = Q + S$
$C_{22} = P + R - Q + U$

Strassen's Matrix Multiplication

In order to improve the algorithm we have to reduce the number of multiplication by introducing the new variables as follows:

- $P = (A_{11} + A_{22}) \times (B_{11} + B_{22})$
- $Q = (A_{21} + A_{22}) \times B_{11}$
- $R = A_{11} \times (B_{12} - B_{22})$
- $S = A_{22} \times (B_{21} - B_{11})$
- $T = (A_{11} + A_{12}) \times B_{22}$
- $U = (A_{21} - A_{11}) \times (B_{11} + B_{12})$
- $V = (A_{12} - A_{22}) \times (B_{21} + B_{22})$

Now, we have:

$C_{11} = P + S - T + V$
$C_{12} = R + T$
$C_{21} = Q + S$
$C_{22} = P + R - Q + U$

So the $T(n)$ can be expressed as:
$T(n) = 7T(n/2) + O(n^2) = \Theta(n^{\log_2 7})$.

Given an integer array of size n. Find the k-smallest number ?

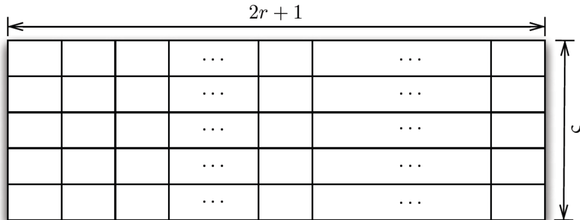
Given an integer array of size n. Find the k-smallest number ?

Suppose $n = 5(2r + 1)$ otherwise we add some zero to the end of the array.

Given an integer array of size n. Find the k-smallest number ?

Suppose $n = 5(2r+1)$ otherwise we add some zero to the end of the array.

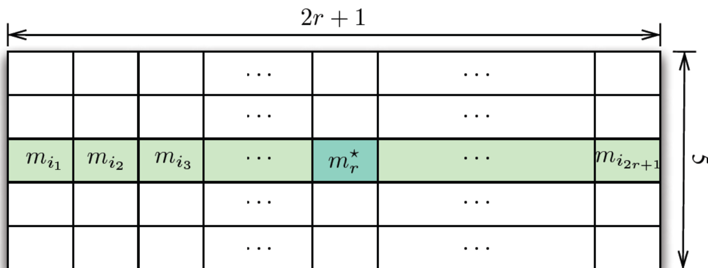
Step 1: Divide n elements into $2r+1$ groups, each of size 5 and arrange them as follows:



This step requires $O(1)$ time complexity.

Find the median for each group.

Step 3: Recursively call the algorithm to find the median of medians:

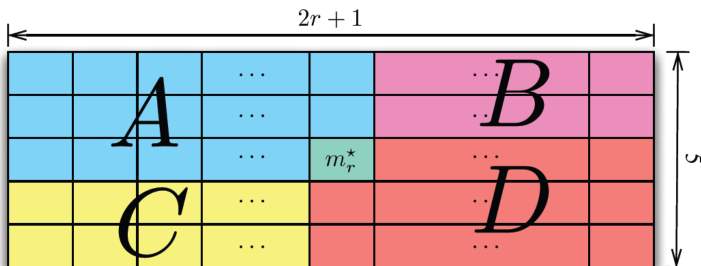


It requires $T(n/5) = T(0.2n)$ time complexity.

Step 4: Constructing the following sets with respect to the value of $m_r^\star$ as follows:

$$L = A \cup \{x \mid x \in B \cup C \;\&\; x \leq m_r^\star\}$$

$$G = D \cup \{x \mid x \in B \cup C \;\&\; x \geq m_r^\star\}$$



This step requires $4r = 0.4n$ comparisons.

Step 5:

- If $|L| = k - 1$ then $return(m_r^\star)$.
- Else if $|L| > k - 1$ then $return(Select(k, |L|))$.
- Else if $|L| < k - 1$ then $return(Select(k - |L| - 1, |G|)$.

Since the number of elements in L or G is at most $3r + 2 + 4r \simeq 7r$, so this step $T(7r) = T(0.7n)$ time complexity. The overall time complexity of this algorithm is as follows:

$$T(n) = 1.6n + T(0.2n) + T(0.7n)$$