از آنجایی که کارای زمانی الگوریتم باید از مرتبه $\Theta(n)$ باشد، مشخص است که تعداد پیمایش های روی لیست باید تعدادی ثابت باشد. الگوریتم بویر–مور با چنین کارایی زمانی و همچنین کارایی فضایی $\Theta(1)$ چنین عمل کرده که با یک بار پیمایش، کاندیدی را با عنوان *عضو غالب احتمالی* مشخص می کند. اما چون ما از وجود عضو غالب در لیست اطمینان نداریم، پیمایشی دیگر در لیست را آغاز کرده و از غالب بودن عنصری که کاندید شده است، اطمینان حاصل می کنیم.

find_candidate(A)

**Require:** a list $A[0, \ldots, n-1]$ containing $n$ objects
**Ensure:** returns a candidate object whose count is possibly more than $\lfloor \frac{n}{2} \rfloor$

```
candidate = 0
votes = 0
for i from 0 to n − 1 do
    if votes = 0 then
        candidate = A[i]
        votes = 1
    else
        if A[i] == candidate then
            votes += 1
        else
            votes -= 1
return candidate
```

الگوریتم فوق، از اولین عضو لیست پیمایش را آغاز می کند. سپس با رویکردی هوشمندانه، به خنثی کردن اعضای لیست می پردازد. بدین صورت که با رسیدن به عنصری مشابه با متغیر $candidate$ ، یک واحد به مقدار متغیر $votes$ می افزاید و در صورت عدم تشابه مقدار متغیر را یک واحد می کاهد. در هر مرحله از پیمایش، در صورتی که مقدار متغیر $votes$ صفر شده باشد، این مفهوم منتقل می شود که تعدادی عضو پیش از این مرحله از پیمایش خنثی شده اند و اکنون نوبت به انتخاب عنصر بعدی به عنوان کاندید رسیده است. در نهایت عنصری که در متغیر $candidate$ باقی می ماند، کاندید و جواب احتمالی ماست.

در واقع عملکرد الگوریتم را می توان به گونه ای دیگر نیز توجیه کرد: ما با آغاز پیمایش، عنصر اول را به عنوان کاندید احتمالی خود در نظر می گیریم. در صورتی که عضو بعدی در لیست مشابه کاندید ما باشد، یک واحد به تعداد رای های کاندید اضافه می کنیم. از طرفی دیگر، در صورت مشاهده عنصری که متفاوت با کاندید ماست،، تعداد آرای کاندید را یک واحد کاهش می دهیم. زمانی که تعداد آراء صفر شود، می توان گفت که پیش از این تعدادی رأی دهنده حضور داشته اند که آرای آنها با یکدیگر خنثی شده است. پس عملا حضور یا عدم حضور آنها در لیست تغییری در برنده یا بازنده شدن کاندید نهایی نخواهد داشت. در نتیجه به بررسی باقی لیست می پردازیم و به دیگر سخن می توان گفت که آرای قبلی را نادیده می گیریم. از این جهت، الگوریتم فوق یک الگوریتم تقلیل و حل به شمار می رود چرا که با هر بار صفر شدن متغیر $votes$ مسأله را به مسأله ای جدید با اندازه $n - (2votes)$ تبدیل می کند.

is_candidate_major(A, candidate)

**Require:** a list $A[0, \ldots, n-1]$ containing $n$ objects, and a *candidate*
**Ensure:** returns candidate if it's a major element, -1 if not

```
count = 0
for i in A do
    if i == candidate then
        count += 1
if count > n // 2 then
    return candidate
else
    return -1
```

الگوریتم فوق نیز، تنها یک بار در لیست پیمایش کرده و تعداد دفعات تکرار عنصر کاندید را در متغیر *count* ذخیره می کند. در نهایت اگر *count* از کف نیمی از تعداد اعضا بیشتر باشد، آن را به عنوان عنصر غالب بر می گرداند. در غیر این صورت $-1$ خروجی داده می شود.

در پایان با ترکیب دو الگوریتم، به پاسخ نهایی می رسیم.

find_major(A)

**Require:** a list $A[0, \ldots, n-1]$ containing $n$ objects
**Ensure:** returns major element if it exists, -1 if not

```
candidate = find_candidate(A)
return is_candidate_major(A, candidate)
```

در اینجا، ابتدا کاندید را یافته و سپس به بررسی غالب بودن آن می پردازیم.

کارایی زمانی الگوریتم فوق $\Theta(2n) = \Theta(n)$ است. به این دلیل که از دو حلقه جداگانه تشکیل شده که هرکدام تمام لیست را پیمایش می کنند. همچنین کارایی فضایی آن نیز به دلیل عدم استفاده از لیستی دیگر به جز لیست ورودی، و صرفا تعدادی ثابت از متغیر ها، $\Theta(1)$ است.