Асимптотичні оцінки

$f(n) \in O(g(n))$

Формальне означення: $\exists n_0 \exists c_{c>0} \forall n_{n>n_0} \big(0 \leqslant f(n) \leqslant c \cdot g(n)\big)$

Приблизний смисл: $f(n) \leqslant g(n)$

$f(n) \in \Omega(g(n))$

Формальне означення: $\exists n_0 \exists c_{c>0} \forall n_{n>n_0} \big(c \cdot g(n) \leqslant f(n)\big)$

Приблизний смисл: $f(n) \geqslant g(n)$

Зв'язок з попередніми: $f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in O(f(n))$

$$f(n) \in \Theta(g(n))$$

Формальне означення:

$$\exists n_0 \exists c_{1 c_1>0} \exists c_{2 c_2>0} \forall n_{n>n_0} \big(c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n)\big)$$

Приблизний смисл: $f(n) \approx g(n)$

Зв'язок з попередніми:

$$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$$

$f(n) \in o(g(n))$

Формальне означення: $\forall c_{c>0} \exists n_0 \forall n_{n>n_0} \big(0 \leqslant f(n) < c \cdot g(n)\big)$

Приблизний смисл: $f(n) < g(n)$

Зв'язок з попередніми:

$f(n) \in o(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge f(n) \notin \Omega(f(n))$

$f(n) \in \omega(g(n))$

Формальне означення: $\forall c_{c>0} \exists n_0 \forall n_{n>n_0} \big(c \cdot g(n) < f(n)\big)$

Приблизний смисл: $f(n) > g(n)$

Зв'язок з попередніми:

$f(n) \in \omega(g(n)) \Leftrightarrow f(n) \in \Omega(g(n)) \wedge f(n) \notin O(f(n))$,

$f(n) \in \omega(g(n)) \Leftrightarrow g(n) \in o(f(n))$

Приклади:

$2021 \in \Theta(1)$

$2n^2 + 1000000n \notin O(n)$

$2n^2 + 1000000n \in O(n^2)$

$2n^2 + 1000000n \in \Theta(n^2)$

$2n^2 + 1000000n \notin o(n^2)$

$2n^2 + 1000000n \in O(n^3)$

$2n^2 + 1000000n \notin \Theta(n^3)$

$2n^2 + 1000000n \in o(n^3)$

$\log_2 n \in o(n)$

$\sqrt{n} \in o(n)$

Наприклад, проаналізуємо детальніше $2021 \in \Theta(1)$.

Якщо міркувати приблизно: ми нехтуємо константними множниками, тому будь-яке конкретне число, зокрема й 2021, замінити на просто 1.

Якщо міркувати формально: треба показати, що $\exists n_0 \exists c_1 {}_{c_1>0} \exists c_2 {}_{c_2>0} \forall n_{n>n_0} \big(c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n)\big)$, де $f(n) = 2021$, $g(n) = 1$, тобто $\exists n_0 \exists c_1 {}_{c_1>0} \exists c_2 {}_{c_2>0} \forall n_{n>n_0} \big(c_1 \cdot 1 \leqslant 2021 \leqslant c_2 \cdot 1\big)$. Ми можемо взяти $n_0 = 1$, $c_1 = 2000$, $c_2 = 2050$, вираз перетворюється у $\forall n_{n>n_0} \big(2000 \cdot 1 \leqslant 2021 \leqslant 2050 \cdot 1\big)$, це правда (доведено).

Ще, наприклад, проаналізуємо $2n^2 + 1000000n \in \Theta(n^2)$.

Якщо міркувати приблизно: ми нехтуємо константними множниками, тому $2n^2$ можна перетворити у n^2, а $1000000n$ можна взагалі відкинути, бо там лише перша степінь n, тоді як в n^2 дру́га. Тобто, від лівої частини лишається n^2, і очевидно, що $n^2 \approx n^2$.

Якщо міркувати щодо того ж $2n^2 + 1000000n \in \Theta(n^2)$ формально: треба показати, що $\exists n_0 \exists c_{1 c_1>0} \exists c_{2 c_2>0} \forall n_{n>n_0} \big(c_1 \cdot g(n) \leqslant \leqslant f(n) \leqslant c_2 \cdot g(n)\big)$, де $f(n) = 2n^2 + 1000000n$, $g(n) = n^2$, тобто $\exists n_0 \exists c_{1 c_1>0} \exists c_{2 c_2>0} \forall n_{n>1} \big(c_1 \cdot n^2 \leqslant 2n^2 + 1000000n \leqslant c_2 \cdot n^2\big)$. Ми можемо взяти, наприклад $n_0 = 100$, $c_1 = 2$, $c_2 = 10002$, вираз перетворюється у $\forall n_{n>100} \big(2 \cdot n^2 \leqslant 2n^2 + 1000000n \leqslant 10002 \cdot n^2\big)$; для цього віднімемо з усіх частин по $2n^2$, лишиться нерівність $0 \leqslant 1000000n \leqslant 10000 \cdot n^2$; поділимо на $10000n$ (воно додатне, тож ділити на нього можна), лишиться нерівність $0 \leqslant 100 \leqslant n$; така нерівність при $n > 100$ виконується.

Звідки в останньому прикладі взялися са́ме $n_0 = 100$, $c_1 = 2$, $c_2 = 10002$? Просто як *деякі* чи́сла, які дають, що треба. Замість них можна взяти й інші. Наприклад, $n_0 = 1$, $c_1 = 2$, $c_2 = 1\,000\,002$ теж підходять, і $n_0 = 10$, $c_1 = 1$, $c_2 = 500\,000$ теж. Сказано "$\exists$" (існує хоча б одне число, таке, що); ми таке число пред'явили (точніше, по одному числу на кожен з трьох кванторів "$\exists$"); отже, задовольнили умову.

Але доведення, звісно, не обмежується тим, щоб назвати будь-які числа навмання. Наприклад, $n_0 = 5$, $c_1 = 2$, $c_2 = 3$ не підходить, бо далеко не при кожному $n > 5$ виконується $\left(2 \cdot n^2 \leqslant 2n^2 + 1000000n \leqslant 3 \cdot n^2\right)$, зокрема при $n = 6$ маємо середню частину $2 \cdot 6^2 + 1000000 \cdot 6 = 6000072$, праву частину $3 \cdot n^2 = 108$, і нерівність не виконалася, й у підсумку нічого не доведено (але й не спростовано, бо в означенні Θ використовується "$\exists$", тож те, що не підходить така трійка, не забороняє шукати інші).

Втім, аналіз алгоритмів рідко потребу́є са́ме розглянутих досі теоретичних означень та викладок. Значно частіше починають не із аналітичного вигляду функції, а з коду програми (чи іншого формулювання алгоритму, але теж досить близького до коду програми) і застосовують такі міркування, як:

- Елементарна дія — значить, $\Theta(1)$.
- Цикл, що повторюється рівно n разів — значить, $\Theta(n)$ (за умови, що кожна окрема ітерація має складність $\Theta(1)$).
- Цикл, що повторюється щонайбільше n разів, але може й менше (наприклад, містить умови дострокового виходу, які при деяких вхідних даних спрацьовують) — значить, $O(n)$ (за тієї ж умови).

- Вкладені цикли, де зовнішній повторюється n разів, внутрішній m разів — значить, $\Theta(m \cdot n)$ (якщо рівно n та рівно m, інакше $O(m \cdot n)$), за тієї ж умови, що складність кожної окремо ітерації внутрішнього цикла $\Theta(1)$.
- Розгалуження, одна гілка якого має складність $O(f(n))$, інша $O(g(n))$ — значить, $O(\max(f(n), g(n)))$.
- Послідовне виконання фрагментів, які мають складності $O(f(n))$ та $O(g(n))$ — формально $O(f(n) + g(n))$, але можна виразити і як $O(\max(f(n), g(n)))$; якщо очевидно, яка з цих функцій більша, то отримуємо відповідно або $O(f(n))$, або $O(g(n))$.

Асимптотична оцінка часу роботи сортування вибором

```
int n = a.Length;
for(int i=n-1; i>0; i--) {
  int i_max = i;
  for(int j=0; j<i; j++)
    if(a[j]<a[i_max])
      i_max = j;
  swap(ref a[i_max], ref a[j]); // обмін місцями
}
```

$\left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} \Theta(1) \Bigg\} \Theta(i)$

Можна сказати, що вартість внутрішнього цикла $\Theta(i)$ являє собою також і $O(n)$, і тоді $\Theta(n)$ разів (ітерацій зовнішнього цикла) по $O(n)$ (вартість внутрішнього цикла) сумарно буде $O(n^2)$. А можна провести трохи детальніші перетворення, щоб побачити, що має місце не лише O, але також і Θ:

$$n + (n-1) + \ldots + 2 = \frac{n(n+1)}{2} - 1 = 0{,}5n^2 + 0{,}5n - 1 \in \Theta(n^2)$$

Асимптотична оцінка часу сортування простими вставками

```
for(int i=1; i<n; i++) { // де n = a.Length
  int curr = A[i];
  int j = i-1;
  while (j>=0 && A[j]>curr) {
    A[j+1] = A[j];
    j--;
  }
  A[j+1] = curr;
}
```

$\Theta(1)$ — тіло внутрішнього циклу; $O(i) \subseteq O(n)$ — тіло зовнішнього циклу.

Сумарно буде $\Theta(n)$ разів по $O(n)$, тобто $O(n^2)$.

Якщо розглядати довільні (а не спеціально вибрані) вхідні дані, виходить са́ме $O(n^2)$, яке неможливо перетворити у Θ від чого б не було: при деяких вхідних даних (наприклад, уже відсортованих) маємо $\Theta(n)$ разів по $\Theta(1)$, тобто $\Theta(n)$; при деяких інших вхідних даних (наприклад, відсортованих у протилежному порядку) маємо $\Theta(n)$ разів по $\Theta(n)$, тобто $\Theta(n^2)$. А $\Theta(n)$ та $\Theta(n^2)$ — несумісні вимоги ($\Theta(n) \cap \Theta(n^2) = \varnothing$)

Неточно, зате характеризується алгоритм, а не програма.

Як використовувати асимптотичні оцінки? Порівнювати ефективність алгоритмів. А також, приблизно оцінювати шанси алгоритму помістити́сь у такі-то обмеження часу при вхідних даних таких-то розмірів. (І для всього цього не завжди треба реалізовувати їх, часто досить уявити структуру циклів.)

Тактова частота сучасних комп'ютерів — кілька гігагерців (мільярдів тактів за секунду). Враховуючи неточності у питанні «скільки тактів займають які дії» та ігнорування констант у самих́ означеннях O чи Θ, ці «кілька мільярдів» треба ще поділити (причому не ясно, на скільки), й у висновку виходить щось дуже приблизне «за секунду встигається десь 10^7–10^9 дій» (причому, навіть це не завжди гарантовано...)

Але навіть настільки приблизні оцінки бувають корисними.

Наприклад: $n = 10^8$, складність алгоритму $\Theta(n^2)$. Тоді $(10^8)^2 = 10^{16}$, час роботи десь від $\frac{10^{16}\text{дій}}{10^9\text{дій/сек}} = 10^7$ сек (≈ 4 місяці) до $\frac{10^{16}\text{дій}}{10^7\text{дій/сек}} = 10^9$ сек (≈ 30 років) — дочекатися нереально.

Або: $n = 10^5$, складність алгоритму $O(n\sqrt{n})$. Тоді ($10^5 \cdot \sqrt{10^5} \approx$ $\approx 3{,}2 \cdot 10^7$), і час роботи не перевищує десь чи то $\frac{3{,}2 \cdot 10^7\text{дій}}{10^9\text{дій/сек}} \approx$ $\approx 0{,}03$ сек, чи то $\frac{3{,}2 \cdot 10^7\text{дій}}{10^7\text{дій/сек}} \approx 3$ сек — дочекатися точно можна.

(Але якщо необхідно, наприклад, вкластися в 1 сек — це не гарантовано. Можуть бути важливі подальші оптимізації (чи то принципові, які зменшать асимптотичну оцінку, чи то дрібні, якими асимптотичний аналіз нехтує, але ж на час вони впливають). Усе це слід з'ясовувати, і враховуючи суть алгоритму, і замірюючи час експериментально; знання само́ї лише асимптотичної оцінки $O(n\sqrt{n})$ тут вже не допоможе.)

Звісно, можна вимірювати час і по-простому в мілісекундах. Автоматичні системи перевірки (включаючи хоч eolymp, хоч ejudge) так і вимірюють. Ці способи не заважають один о́дному, а доповнюють.

Асимптотичні оцінки дуже помічні для питання *«як зміниться час роботи, якщо змінити розмір вхідних даних?»*. Наприклад: *«Скільки часу оброблятимуться 5000 елементів, якщо 1000 елементів оброблялися 1 хвилину?»*. Можна, звісно, голослівно заявити, ніби «уп'ятеро більші вхідні дані — значить, уп'ятеро більший час»... Тільки це часто неправда. Точніше, це правда, *лише якщо складність алгоритму близька до* $\Theta(n)$. А якщо складність, наприклад, $\Theta(n^3)$, то збільшення вхідних даних у 5 разів збільшує час у $5^3 = 125$ разів... (Приблизно! Доданки, якими нехтує асимптотика, можуть перетворити 125 у 80 чи у 150... але 5^3 — все ж непогане наближення, значно краще, чим 5.) (приклад!)

Аналогічно можна розглянути обернену задачу: *«наскільки можна збільшити розмір вхідних даних, якщо дозволена кількість операцій збільшилася так-то?»*.

Припустимо, на старому комп'ютері реально дочекатися, доки виконаються 10^7 операцій, а на новому 10^{10}. Обробку задачі якого розміра тоді реально дочекатися? Наскільки зросте цей розмір внаслідок заміни старого компа на новий?

Припустивши, ніби мова не про асимптотику, а про точні функції, маємо:

кількість дій	старий комп	новий комп	
n	10^7	10^{10}	у 1000 разів
$n \log_2 n$	62 тис.	39,6 млн.	у 631 раз
$n\sqrt{n}$	46 тис.	4,6 млн.	у 100 разів
n^2	3,16 тис.	100 тис.	у 31 раз
n^3	215	2154	у 10 разів
2^n	23	33	на 10
$n!$	10	13	на 3

Якщо маємо асимптотику, а не точні функції, ці числові результати неточні. Але різниця між, наприклад, «у 631 раз для $n \log_2 n$» та «у 10 разів для n^3» більша за ці неточності, й загальна картина са́ме така: чим гірша (більша) асимптотична оцінка, тим менший приріст розміру вхідних даних від того самого збільшення потужності комп'ютера.

Нарешті, в асимптотичному аналізі часто використовують записи $O(n \log n)$, $\Theta(\log n)$, тощо, в яких *не* вказана основа. Це не помилка, а традиція.

По-перше, завдяки тотожності

$$\log_a n = \log_a b \cdot \log_b n,$$

вирази $O(\log_a n)$ та $O(\log_b n)$ при довільних константних $a > 1$, $b > 1$ задають один і той самий клас функцій (бо $\log_a b$ виявляється константою, якою асимптотичний аналіз нехтує).

По-друге, у комп'ютерних науках «стандартною» основою логарифма вважається 2, тож $\log n$ можна трактувати як $\log_2 n$.

Груповий метод аналізу алгоритмів

Розглянемо задачу: *«Є гарантовано відсортований за строгим зростанням масив* `int[] A = new int[n]`. *Крім нього, задане число* `SUM_NEED`. *Скільки є різних способів задати це число як суму рівно двох значень з масиву? Допускається (якщо так виходить потрібна сума), щоб це був двічі взятий один елемент. Способи* `a[i]+a[k]` *та* `a[k]+a[i]` *(при* $i \neq k$*) вважати різними.»*

Найпростіший розв'язок — десь такий:

```
int res = 0;
for(int i=0; i<n; i++)
  for(int j=0; j<n; j++)
    if(A[i]+A[j] == SUM_NEED)
      res++;
```

$\Theta(1)$, $\Theta(n)$, $\Theta(n\hat{}2)$

Завдяки відсортованості масиву, можливий бінарний пошук:

```
int res = 0;
for(int i=0; i<n; i++) {
  <спробувати знайти SUM_NEED - A[i] в A бін.пошуком>
  if(<результат бін.пошуку успішний>)
    res++;
}
```

Як відомо, час роботи бінарного пошуку становить $O(\log n)$; отже, фрагмент у цілому (*після* правильної реалізації бінарного пошуку мовою програмування) має складність $O(n \log n)$.

$\log n \in o(n)$ — отже, $O(n \log n)$ значно менше, чим $\Theta(n^2)$.

Але саме в цій задачі *най*ефективніший підхід несподівано *не* потребує бінарного пошуку, а схожий на найпростіший підхід. Перепишемо простий підхід так (↓), потім помітимо кілька дрібних фактів:

```
int res = 0;
for(int i=0; i<n; i++) {
  int j = n-1;
  while(j>=0) {
    if(A[i]+A[j]==SUM_NEED)
      res++;
    j--;
  }
}
```

(1) раз масив відсортований і нам треба `A[i]+A[j] == SUM_NEED`, то смисл зменшувати `j` (при незмінному `i`) є лише поки $A[i] + A[j] > SUM\_NEED$.
Отже, `while(j>=0)` можна змінити на `while(j>=0 && A[i]+A[j]>SUM_NEED)`.
(2) якщо при деякому $\texttt{i} = i^\star$ умова `A[i]+A[j] <= SUM_NEED` була досягнута при $\texttt{j} = j^\star$ (при $\texttt{j} = j^\star + 1$ ще було `A[i]+A[`$j^\star + 1$`] > SUM_NEED`), то при наступному $\texttt{i} = i^\star + 1$ можна починати з цього самого $\texttt{j} = j^\star$, а не заново з `j=n-1`.

```
int res = 0;
int j = n-1;
for(int i=0; i<n; i++) {
  while(j>=0 && A[i]+A[j]>SUM_NEED)
    j--;
  if(A[i]+A[j]==SUM_NEED)
    res++;
}
```

Аналізуючи цей алгоритм так само, як аналізували сортування вставками, отримаємо те ж $O(n^2)$. Поки що не ясно, чому заявлено, ніби це краще, чим бінпошук за $\Theta(n \log n)$.

```
int res = 0, j = n-1;
for(int i=0; i<n; i++) {
  while(j>=0 && A[i]+A[j]>SUM_NEED)
    j--;  // цей while СУМАРНО ПО ВСІМ ІТЕРАЦІЯМ for-а
// займе O(n) часу, бо j лише зменшується від n-1
// не далі як до 0. Отакі міркування і називають
  if(j>=0 && A[i]+A[j]==SUM_NEED) // груповим аналізом
    res++;
}
```

Але насправді тут $\Theta(n)$, просто щоб це показати, треба вжити так званий *груповий аналіз*: вкладений цикл аналізується *сумарно по всім* ітераціям зовнішнього циклу.

До речі, сам прийом з i та j називають *два вказівники* (два указателя, two pointers).

Оцінки складності *задачі*

Верхня оцінка складності задачі — найнижча серед складностей усіх відомих алгоритмів, що її розв'язують. Тобто, складність найкращого (найшвидшого) з відомих алгоритмів.

Нижня оцінка складності задачі — математично доведене твердження, що ніякий, *ні вже відомий, ні ще не відомий* алгоритм, який правильно й повно розв'язує задачу, не може мати складність меншу, чим вказана.

Тривіальна нижня оцінка складності задачі — оцінка часу, потрібного на прочитання вхідних даних і виведення результату.

Зазвичай, алгоритм не може працювати правильно, не читаючи вхідні дані або не виводячи результат. Тому загальний час роботи алгоритма зазвичай включає в себе час на читання даних і виведення результату.

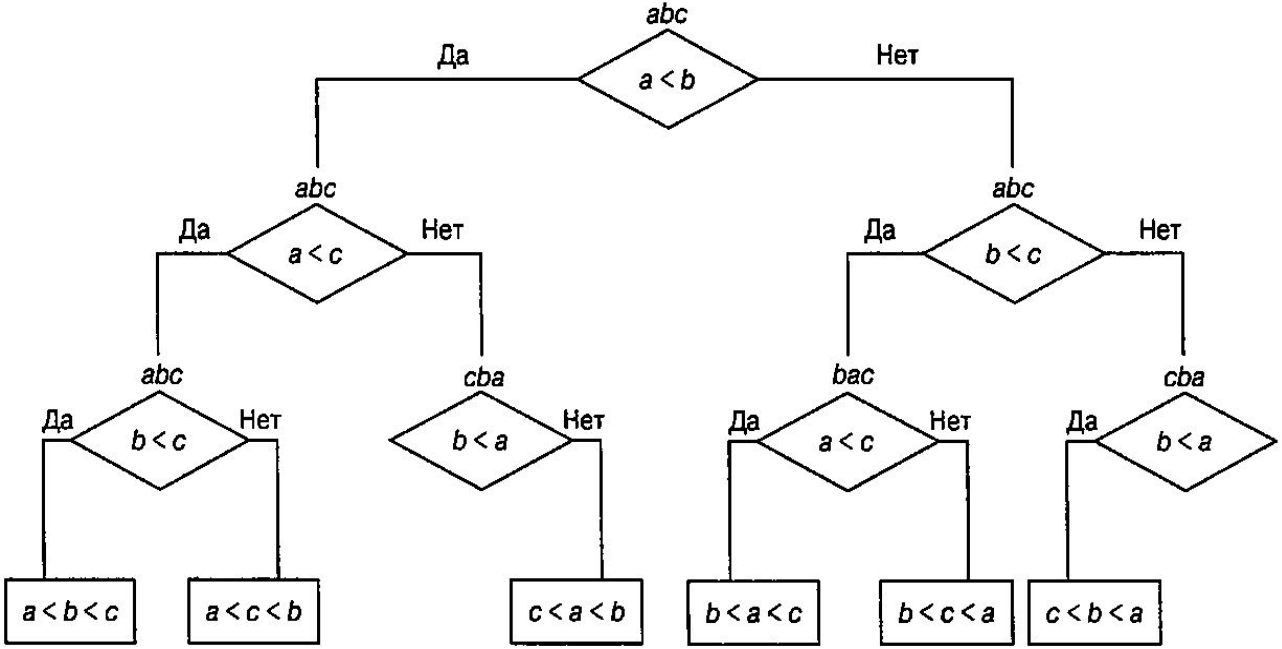
Але в деяких окремих ситуаціях можливі й виключення, коли тривіальна нижня оцінка може втрачати смисл. Наприклад:

(А) Задача поставлена дивно, і вхідні дані в повному обсязі насправді не потрібні (наприклад: *«Дано n чисел. Чи від'ємне перше з них?»*).

(Б) При застосуванні алгоритма як *під*задачі, виявляється, що потрібні дані вже розміщені у пам'яті потрібним чином. Наприклад, у одному з раніше згаданих алгоритмів пропонувалося багатократно шукати у відсортованому масиві `SUM_NEED - A[i]`. *Якби* для кожного пошуку доводилося заново читати/формувати масив, у якому робиться пошук, оцінка кожного з таких пошуків *була б* $\Theta(n)$ са́ме як тривіальна оцінка на читання даних. Але всі ці пошуки виконуються в уже готовому масиві, тому кожен з цих пошуків (якщо зробити його бінарним) «коштує» лише $O(\log n)$.

Доведення нижньої оцінки складності задачі сортування шляхом аналізу дерев рішень

Приклад де́рева рішень:



- Час роботи $T(n)$ може бути приблизно виражений як кількість порівнянь елементів = кількість ярусів дéрева рішень.
- Треба вміти розрізнити усі $n!$ можливих вхідних даних.[1]
- Бінарне дерево з максимальною кількістю ярусів k може містити щонайбільше 2^k елементів.

З усього цього слідує: яке б не було дерево рішень (тобто, який би не був алгоритм цих порівнянь), для розрізнення усіх можливих ситуацій треба $2^{T(n)} \geqslant n!$.

[1]Припускаємо, що сортується послідовність попарно різних чисел: тоді всі можливі вхідні дані для однієї й тієї ж відповіді являють собою в точності престановки, а їх $n!$. Від урахування того, що елементи можуть бути не лише всі різні, а ще й можуть бути частково однакові, кількість способів і час роботи явно не стануть меншими, так що *нижня* оцінка лишається справедливою.

Прологарифмувавши $2^{T(n)} \geqslant n!$, отримуємо $T(n) \geqslant \log_2(n!)$.

За формулою Стірлінга, $\lim\limits_{n\to\infty} \dfrac{n!}{\left(\frac{n}{e}\right)^n \cdot \sqrt{2\pi n}} = 1$ (де $e \approx 2{,}71828$ — основа натурального логарифму), звідки $n! = \Theta\left(\left(\dfrac{n}{e}\right)^n \cdot \sqrt{2\pi n}\right)$.

$$
\begin{aligned}
\text{Звідси, } \log_2(n!) &\approx \log_2\left(\left(\tfrac{n}{e}\right)^n \cdot \sqrt{2\pi n}\right) = \\
&= \log_2\left(\left(\tfrac{n}{e}\right)^n\right) + \log_2\sqrt{2\pi n} = \\
&= n \cdot (\log_2 n - \log_2 e) + \tfrac{1}{2} \cdot (\log_2 2\pi + \log_2 n) = \\
&= n\log_2 n - n\log_2 e + \tfrac{1}{2}\big(\log_2 n + \log_2 2\pi\big) \in \\
&\in \Theta(n\log_2 n).
\end{aligned}
$$

Є й простіше доведення, що не потребує формули Стірлінга. Розпишемо факторіал як добуток всіх чисел від 1 до n (розглядаємо лише додатні n, тому так можна), і замінимо кожен з перших $(n \operatorname{div} 2)$ множників на 1, а кожен з решти на $n/2$:

$$\underbrace{1 \cdot 2 \cdot \ldots \cdot (n \operatorname{div} 2)}_{\text{замінимо кожен на 1}} \times \underbrace{(n \operatorname{div} 2 + 1) \cdot (n \operatorname{div} 2 + 2) \cdot \ldots \cdot n}_{\text{замінимо кожен на } n/2} >$$

$$> \underbrace{1 \cdot 1 \cdot \ldots \cdot 1}_{(n \operatorname{div} 2) \text{ штук}} \times \underbrace{(n/2) \cdot (n/2) \cdot \ldots \cdot (n/2)}_{(n - n \operatorname{div} 2) \text{ штук}} =$$

$$= 1^{n \operatorname{div} 2} \times (n/2)^{n - n \operatorname{div} 2} \geqslant (n/2)^{n/2}.$$

(Щоб довести це цілком строго, слід розглянути окремо випадки па́рного та непа́рного n, і для кожного з них показати, що $n \operatorname{div} 2 + 1 \geqslant n/2$ та $n - n \operatorname{div} 2 \geqslant n/2$, але за́раз це пропустимо.)

Прологарифмуємо:

$$\log_2((n/2)^{n/2}) = \frac{n}{2}\log_2\frac{n}{2} = \frac{n}{2}\cdot(\log_2 n - 1).$$

Враховуючи $(n/2) \in \Theta(n)$ та $(\log_2 n - 1) \in \Theta(\log n)$, маємо $\log_2((n/2)^{n/2}) \in \Theta(n\log n)$, а поєднавши це з $n! > (n/2)^{n/2}$, отримуємо $\log_2 n! \in \Omega(n\log n)$.

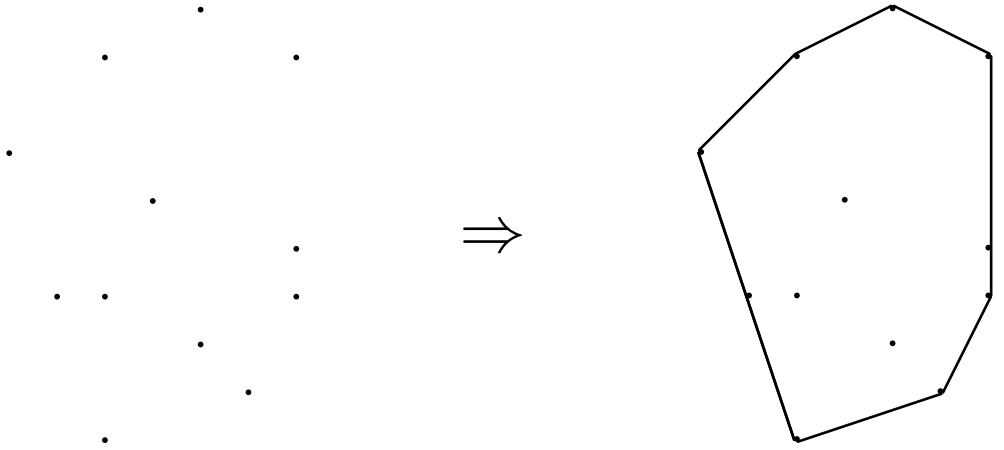
Це доведення гірше, чим через формулу Стірлінга, бо тоді довели Θ, а тепер лише Ω. Однак, оскільки далі це все'дно використовується для нижньої оцінки, це не важливо.

Дрібне бонусне завдання 1. Виконайте строге доведення тверджень $n \operatorname{div} 2 + 1 \geqslant n/2$ та $n - n \operatorname{div} 2 \geqslant n/2$.

Дрібне бонусне завдання 2. Запропонуйте, як засобами, аналогічними дру́гому доведенню (але простішими), довести також і $\log_2 n! \in O(n\log n)$ (що разом з уже доведеним дасть $\log_2 n! \in \Theta(n\log n)$).

Доведення нижньої оцінки складності шляхом зведення (на прикладі задачі опуклої оболонки у площині)

Постановка задачі побудови опуклої оболонки (для двовимірного випадку): на площині задано (в довільному порядку) сукупність точок; знайти мінімальний опуклий многокутник, який повністю їх включає.



Доведемо, що ця задача має складність $\Omega(n \log n)$.

Припустимо, ніби існує алгоритм її розв'язання, що має складність $o(n \log n)$. Цей гіпотетичний алгоритм читає вхідні дані (сукупність пар координат (x_i, y_i) у довільному порядку) й виводить вершини опуклої оболонки в порядку обходу (наприклад, проти годинникової стрілки).

Тоді було б можливо розробити новий алгоритм сортування. Щоб відсортувати N чисел $a_1, a_2, \ldots, a_N$:

1. перетворимо їх у точки з координатами $(a_1, a_1^2), (a_2, a_2^2), \ldots, (a_N, a_N^2)$;
2. застосуємо той алгоритм побудови опуклої оболонки;
3. виведемо послідовність x-координат точок, від найлівішої до найправішої.

   (Раз ми маємо многокутник, в порядку обходу, то для цього їх треба виводити просто підряд. У крайньому разі, якщо сформований гіпотетичним алгоритмом многокутник починається не з крайньої лівої точки, або порядок обходу за годинниковою стрілкою, а не проти — не виключено, що доведеться виводити два різні шматки, або у зворотньому порядку... Але все це *можливо*.)

Складність усіх етапів, крім виклику алгоритму побудови опуклої оболонки, $\Theta(n)$, що $o(n \log n)$. Сам алгоритм побудови опуклої оболонки, за припущенням, теж $o(n \log n)$. Таким чином, складність усього сортування теж $o(n \log n)$.

Але ми знаємо доведення, що це не так!

Таким чином припущення, ніби існує алгоритм побудови опуклої оболонки, що має складність $o(n \log n)$, призвело до протиріччя. Значить, насправді такого алгоритму побудови опуклої оболонки не існує.

У практичному програмуванні: щоб придумати новий розв'язок, намагаємося використати як складову відомий алгоритм розв'язування відомої задачі.

При доведенні нижньої оцінки методом зведення, навпаки, намагаємося використати невідомий, гіпотетичний алгоритм розв'язування нової задачі як складову алгоритму розв'язування відомої задачі, для якої знаємо нижню оцінку. (І поєднуємо це з класичною ідеєю доведення від супротивного.)

Якщо побудувати таке перетворення вдається, цим доводиться, що нова задача має не нижчу, чим відома, нижню оцінку складності.

«Парадокс» сортування підрахунком

Нехай треба відсортувати масив `unsigned short` (беззнакових 16-бітових чисел, вони ж `UInt16`). Це можна зробити так:

```
int[] num = new int[0x10000];
for(int i=0; i<data.Length; i++) {
    num[data[i]]++;
}
for(long v=0, i=0; v<=0xFFFF; v++) {
    for(int q=0; q<num[v]; q++, i++) {
        data[i] = (UInt16)v;
}
```

} Цей алгоритм називається *сортування підрахунком* (рос. сортировка подсчётом; англ. counting sort або bucket sort).

Наприклад, відсортуємо цим алгоритмом масив (2, 5, 4, 7, 2, 12, 4, 1, 2). З нього буде отримано масив `num` вигляду (0, 1, 3, 0, 2, 1, 0, 1, 0, 0, 0, 0, 1, ще 65523 штук “0”). Потім відбудеться очистка старого масиву `data` й формування його по-новій:

`num[0]` = 0, тому нулів не додаватимуть;

`num[1]` = 1, тому буде додана одна штука “1”;

`num[2]` = 3, тому буде додано три штуки “2”;

`num[3]` = 0, тому трійок не додаватимуть;

`num[4]` = 2, тому буде додано дві штуки “4”;

і так далі.

Неважко (особливо знаючи про груповий аналіз) переконатися, що сортування підрахунком (усе в цілому) має складність $\Theta(N + 2^{16}) = \Theta(N)$ (адже $2^{16} \in \Theta(1)$), де $N = \texttt{data.Length}$.

А це протирічить раніше доведеній нижній оцінці задачі сортування. У цьому і полягає «парадокс»: спочатку довели, що швидше, чим за $n \log n$, ніяк не можна, а потім пред'явили алгоритм, що працює за $\Theta(n) \subset o(n \log n)$.

Є щонайменше дві причини, чому це все-таки не справжнє протиріччя і не справжній парадокс.

1) Так можна відсортувати масив `unsigned short`-ів, але не масив, наприклад, `double`-ів. Сортування підрахунком *менш універсальне*, чим класичні алгоритми сортування. А ситуація «для простішої задачі існує ефективніший, чим для складнішої, алгоритм» абсолютно ніякого протиріччя не містить.

2) Доведення через дере́ва рішень містить *припущення*, що алгоритм формує результат, спираючись на результати порівнянь («якщо `a[i]<a[j]`, то зробити одне, інакше інше»). А сортування підрахунком не спирається на порівняння, тому на нього це доведення і не повинно поширюватися.

Теоретично, це означає гіпотетичну можливість існування якогось алгоритму, який і сортує елементи довільних типів, і, не спираючись на порівняння, працює швидше $n \log n$. Але незрозуміло, *як* можна одночасно і вміти сортувати елементи довільних типів, і не спиратися на порівняння... І абсолютно точно, що переважна більшість практичних алгоритмів сортування дозволяють задавати компаратор, а отже спираються на порівняння, і для них доведення нижньої оцінки $\Omega(n \log n)$ правильне.